INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

The Design of a 3D-Object Simulator Using Open-GL

Shangbin Zou

A Major Report
In
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

May 2001

Abstract

The Design of a 3D-Object Simulator Using Open-GL

Shangbin Zou

This project analyzes characteristics of 3D objects in the real world, designs a framework by means of object-oriented methodology, which may be used to construct and simulate 3D world easily. Some algorithms such as collision detection and response, boundary check are discussed.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Peter Grogono, for his kind agreement to my initial motivation and proposal. Subsequently his enthusiastic support and valuable guidance gave me an excellent chance to explore the state of the art technology for this project. Without his kind support I could not finish it.

Second I would like to express my sincere thanks to my partner, Ms. Ping Ma, for her contribution to this project -- described in the implementation of a 3D-Object Simulator.

Finally, I would like to thank all people who provide helps and supports for my project.

Contents

1. Introduction

This project demonstrates a prototype of a simple 3D objects simulation coded using Visual C++ Version 6 and OpenGL on Windows environment. The purpose of this project is to design and implement a framework of 3D objects using object-oriented modeling and design methodology; simulate behavior of 3D objects; and show a virtual world of objects. This report describes the design of 3D objects simulation and another report "The Implementation of a 3D-Object Simulator Using Open-GL" [referring to the major report of Ms. Ping Ma] describes the implementation part of this project.

1.1 Aim of the Project

This project is about a simple 3D objects simulation. The aim of this project is to present an Object-Oriented approach to simulate interactions among 3D objects. Our intention is to design and implement a simulation environment that is as realistic as possible. A graphics user interface is provided so that a user can use it to set up initial simulation parameters, control simulation processing and view the simulation result.

1.2 Objectives

In our project, we use object-oriented programming (OOP) for the following reasons:

- It provides us with experience in OOP techniques.
- For 3D objects simulation, it can take advantage of the strengths of object orientation, objects should usually represent real-world things.
- Build up a framework, easy to expand this system.

Since this program is intended to be an extendable framework, we think there are phase objectives and if this goes well there should be further phases.

- Allow people, who are not experts in 3D work, to create and manipulate interesting 3D worlds.
- The user should be able to create and manipulate simple shapes like cube, cone, sphere.
- The system should be able to simulate simple behaviors such as objects moving, objects rotating, objects colliding, etc.
- The program should use existing open standards such as: OpenGL and other storage and rendering standards.
- The system should be easily expandable by extending classes at build time. It should allow the addition of:
  - New shapes
  - New storage formats
  - New behaviors
  - Performance should be good enough for simple simulations.

1.3 Organization of the Report

This report is divided into five chapters. Chapter one is an introduction. It briefly describes the aim and objective of this project. Chapter two describes the background and all concepts of the project. Chapter three presents the analysis and design phase, including detail descriptions for each class. Chapter four gives four simulation scenarios,

each one presents a simulation result. The last chapter concludes the report and suggests further work for the project.

2. Theory behind Project

This chapter describes the background knowledge necessary for designing and implementing the 3D objects simulation.

2.1 Object-Oriented Programming

Object-oriented programming has become the dominant programming style in the software industry over the last ten years or so. The reason for this has to do with the growing size and scale of software projects. It becomes extremely difficult to understand a procedural program once it gets above a certain size. Object-oriented programs scale up better, meaning that they are easier to write, understand and maintain than procedural programs of the same size.

Object-oriented programming appeals at multiple levels. For managers, it promises faster and cheaper development and maintenance. For analysts and designers, the modeling process becomes simpler and produces a clear, manageable design. For programmers, the elegance and clarity of the object model and the power of object-oriented tools and libraries makes programming a much more pleasant task, and programmers experience an increase in productivity. [JMWFW97]

Object-oriented programs are organized around data. The three principles of object-oriented programming are: encapsulation, polymorphism, and inheritance.

2.1.1 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of program that access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. [JMWFW97]

For all intents and purpose, an object is a variable of a user-defined type. Each time we define a new type of object, we are creating a new data type. Each specific instance of this data type is a compound variable.

2.1.2 Polymorphism

Object-oriented programming languages support polymorphism, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. A real-world

example of polymorphism is a thermostat. No matter what type of furnace our house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) we have. For example, if we want a 70-degree temperature, we set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat.

This same principle can also apply to programming. For example, we might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, we can define one set of names, push() and pop(), that can be used for all three stacks. In our program we will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack -- the function push() and pop() -- are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data.

Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. The programmer doesn't need to do this selection manually.

2.1.3 Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. If we think about it, most knowledge is made manageable by hierarchical classifications. For example, a red apple is part of the classification apple, which in turn is part of fruit class, which is under the large class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Inheritance is an important aspect of object-oriented programming.

2.2 3D Graphics Fundamentals

Before getting into our project, we give the fundamental concepts of 3D graphics and coordinate system.

2.2.1 3D Perception

"3D Computer graphics" is actually two-dimensional images on a flat computer screen that provide an illusion of depth, or a third "dimension." In order to truly see in 3D, we need to actually view the object with both eyes, or supply each eye with separate and unique images. Each eye receives a two-dimensional image that is much like a temporary photograph on the retina. These two images are slightly different because they are

received at two different angles. The brain then combines these slightly different images to produce a single, composite 3D picture in our head, as shown in Figure 2-1.

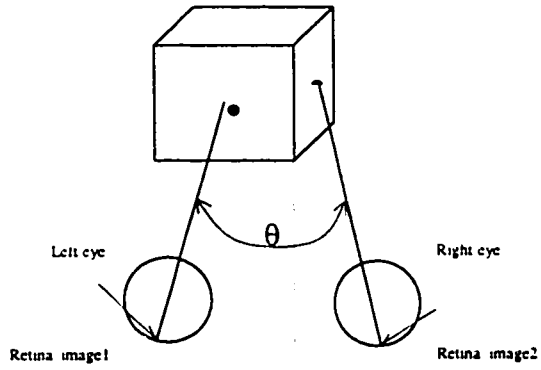

Figure 2.1 How the eyes "see" three dimensions

2.2.1.1 2D + Perspective = 3D

Why doesn't the world suddenly flatten when we cover one eye? The reason is that many of a 3D world's effects are also present in a 2D world. This is just enough to trigger our brain's ability to discern depth. The most obvious cue is that nearby objects appear larger than distant objects and parallel lines do not appear parallel, etc. This effect is called *perspective*. Figure 2-2 presents a simple wire frame cube. [Rsw96]

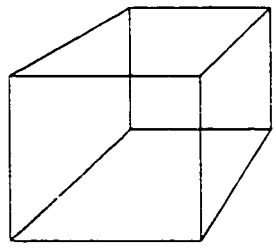

Figure 2-2 This simple wire frame cube demonstrates perspective.

2.2.1.2 Hidden Line Removal

A wire frame cube like figure 2-2 contains enough information to lend the appearance of three dimensions, but not enough to let us discern the front of the cube from the back. When viewing a real object, how do tell the front from the back? Simple – the back is obscured by the front. To simulate this in a two-dimensional drawing, lines that would be obscured by surfaces in front of them must be removed. This is called hidden line removal, as shown in figure 2-3.

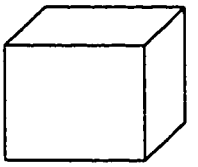

Figure 2-3 The cube after hidden lines are removed

2.2.1.3 Colors and Shading

Figure 2-3 still doesn't look much like a real-world object. The faces of the cube are exactly the same as the background. A real cube would have some color and / or texture. Unless we specifically draw the edges in a different color, there is no perception of three dimensions at all, as shown in Figure 2-4. In order to regain the perspective of a solid object, we need to either make each of the three visible sides a different color, or make them the same color with shading to produce the illusion of lighting.

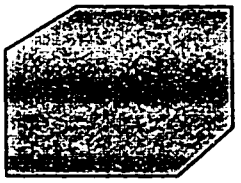

Figure 2-4 The cube with color, but no shading

2.2.1.4 Lights and Shadows

Lighting has two important effects on objects viewed in three dimensions. First, it causes a surface of a uniform color to appear shaded when viewed or illuminated from an angle. Second, objects that do not transmit light cast shadows when they obstruct the path of a ray of light, as shown in figure 2-5.

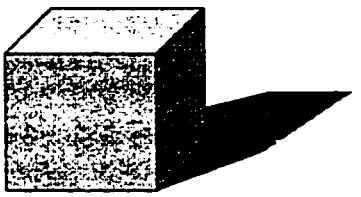

Figure 2-5 A solid cube illuminated by a single light

Two sources of light can influence our three-dimensional objects. Ambient light, which is undirected light, is simply a uniform illumination that can cause shading effects on objects of a solid color; ambient light causes distant edges to appear dimmer. Another kind of light is from a light source, called a lamp. Lamps can be used to change the shading of solid objects and shadow effects.

2.2.2 Coordinate Systems

When drawing points, lines, or other shapes on a computer screen, we usually specify a position in terms of a row and column. If a standard VGA screen has 640 x 480 pixels, the middle of screen should be plotted at (320,240)- that is, 320 pixels from the left of the screen and 240 pixels down from the top of the screen.

2.2.2.1 2D Cartesian Coordinates

The most common coordinate system for two-dimensional plotting is the *Cartesian* system. Cartesian coordinates are specified by an x-coordinate and a y-coordinate. The x-coordinate is a measure of position in the horizontal direction and y is a measure of position in the vertical direction.

The origin of the Cartesian system is at x=0, y=0. Cartesian coordinates are written as coordinate pairs, in parentheses, with the x-coordinate first and the y-coordinate second, separated by a comma. The x and y lines with marks are called the axes and can extend from negative to positive infinity. The x-axis and y-axis are perpendicular and together define the xy plane, as shown in figure 2-6. [Rsw96]

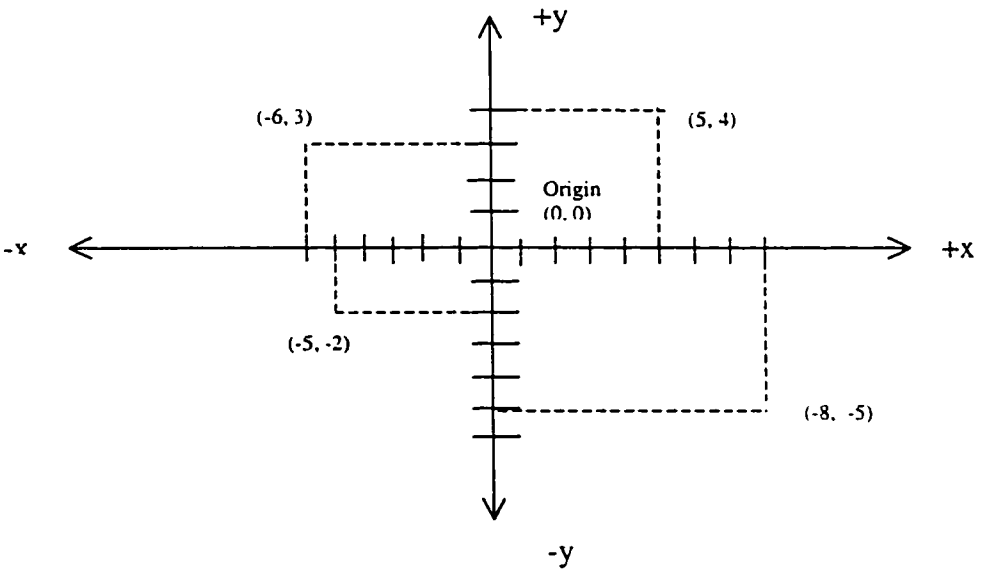

Figure 2-6 The Cartesian plane

2.2.2.2 Coordinate Clipping

A window is measured physically in term of pixels. Before plotting any shapes in a window, we must translate specified coordinate pairs into screen coordinates. This is done by specifying the region of Cartesian space that occupies the window; this region is known as the clipping area. In two-dimensional space, the clipping area is the minimum and maximum x and y values that are inside the window. Figure 2-7 shows a common clipping area, x-coordinates in the window range left to right from 0 to +150, and y-coordinates range bottom to top from 0 to +100. A point in the middle of the screen would be represented as (75,50).

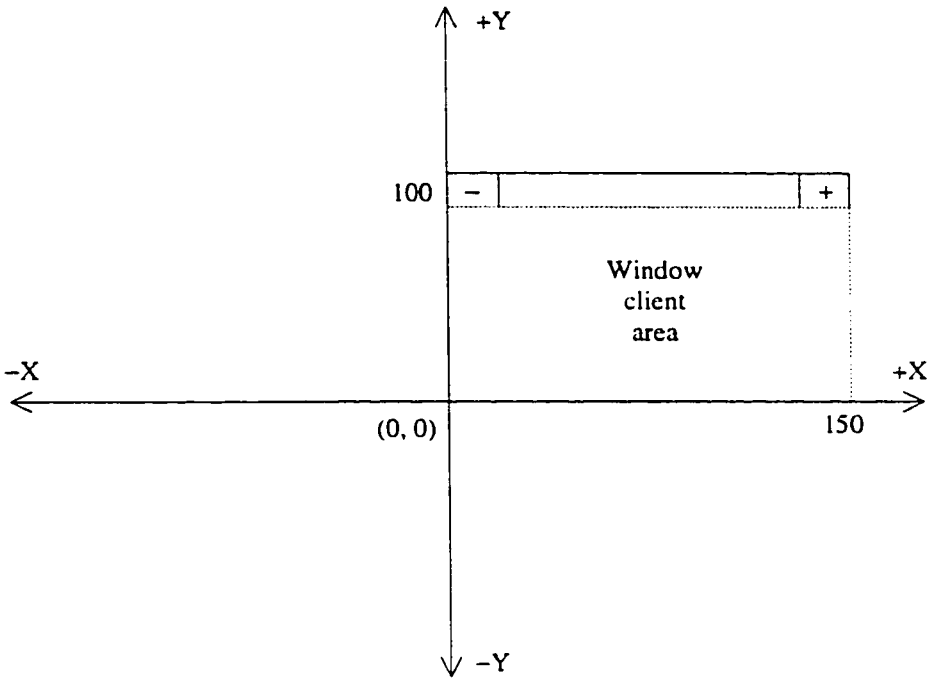

Figure 2-7 One clipping area

2.2.2.3 Viewports

Rarely will a clipping area width and height exactly match the width and height of the window in pixels. The coordinate system must therefore be mapped from logical Cartesian coordinates to physical screen pixel coordinates. This mapping is specified by a setting known as the viewport. The viewport simply maps the clipping area to a region of the window. Usually the viewport is defined as the entire window, as shown in Figure 2-8.

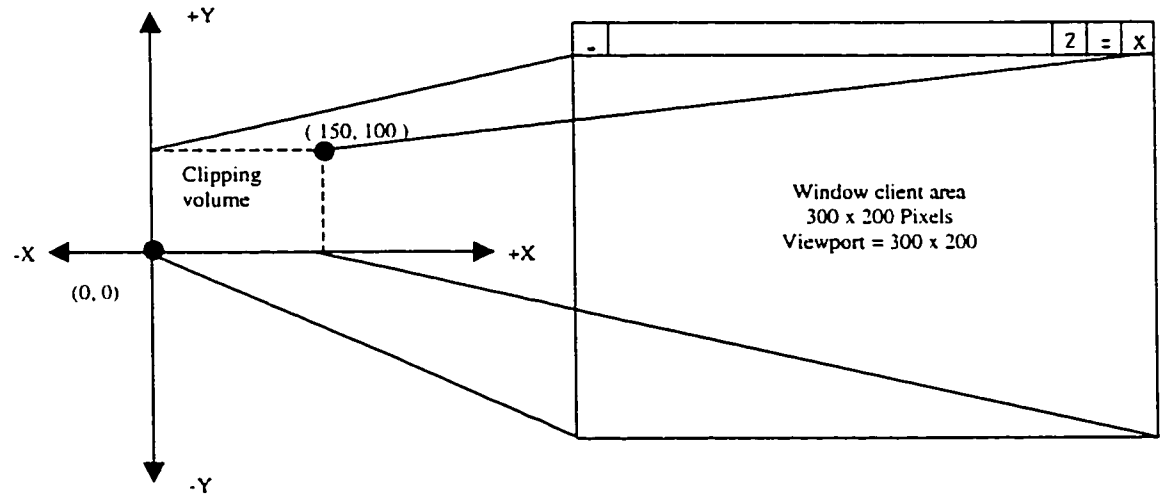

Figure 2-8 A viewport

2.2.2.4 Drawing Primitives

When drawing an object, we actually compose it with several smaller shapes called *primitives*. Primitives are two-dimensional surfaces such as points, lines, and polygons that are assembled in 3D space to create 3D objects. A three-dimensional cube is made up of six two-dimensional squares, each placed on a separate face. Each corner of the square is called a vertex. These vertices are then specified to occupy particular coordinates in 2D or 3D space.

2.2.2.5 3D Cartesian Coordinates

A three-dimensional coordinate system adds a new axis z to the two-dimensional coordinate system. The z-axis is perpendicular to both the x- and y-axes. It represents a line drawn perpendicularly from the center of the screen heading toward the viewer, as shown in figure 2-9.

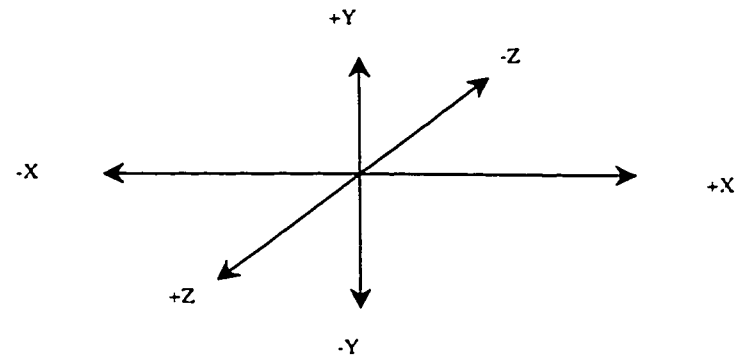

Figure 2-9 Cartesian coordinates in three dimensions

2.2.3 Projections

A mapping of 3D coordinates onto a 2D surface is called a projection. The two main types of projections are orthographic and perspective projections.

2.2.3.1 Orthographic Projections

This projection specifies a square or rectangular clipping volume. Anything outside this clipping area is not drawn. Furthermore, all objects that have the same dimensions appear the same size, regardless of whether they are far away or nearby. This produces a parallel projection, which is useful for drawings of specific objects that do not have any foreshortening when viewed from a distance. A clipping volume in an orthographic projection is defined by specifying the far, near, left, right, top, and bottom clipping planes, as shown in figure 2-10.

In order to view a three-dimensional object on a two-dimensional screen, we must project it. For orthographics projections, the simplest kind of projection simply ignores one

coordinate, the transformation $[x, y, z, 1]^T \rightarrow (x, y)$ provides a screen position for each point by directly ignoring its z coordinate. Here, (x , y, z) defines a point.

Orthographic projections do not give a strong sensation of depth because the size of an object does not depend on its distance from the view. An orthographic projection simulates a view from an infinite distance away.

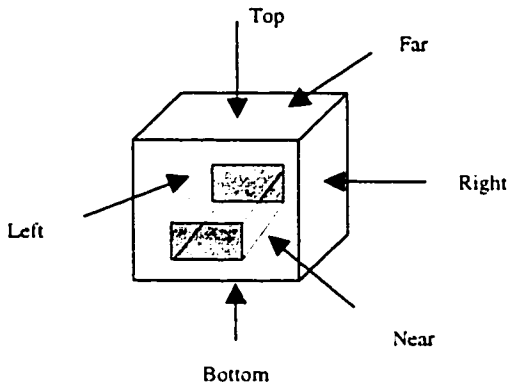

Figure 2-10 The clipping volume for an orthographic projection

2.2.3.2 Perspective Projections

A more common projection is the perspective projection. This projection adds the effect that distant objects appear smaller than nearby objects. The viewing volume as shown in figure 2-11 is something like a pyramid with the top shaved off. This shaved off part is called the frustum. The top and bottom of the frustum determine the distances of the nearest and furthest points that are visible. The sides of the frustum determine the visibility of objects in the other two dimensions. If we think of the closer vertical plane as the screen, the closest objects lie in the plane of the screen and other objects lie behind it.

Objects nearer to the front of viewing volume appear close to their original size, while objects near the back of the volume shrink as they are projected to the front of the volume. This type of projection gives the most realism for simulation and 3D animation.

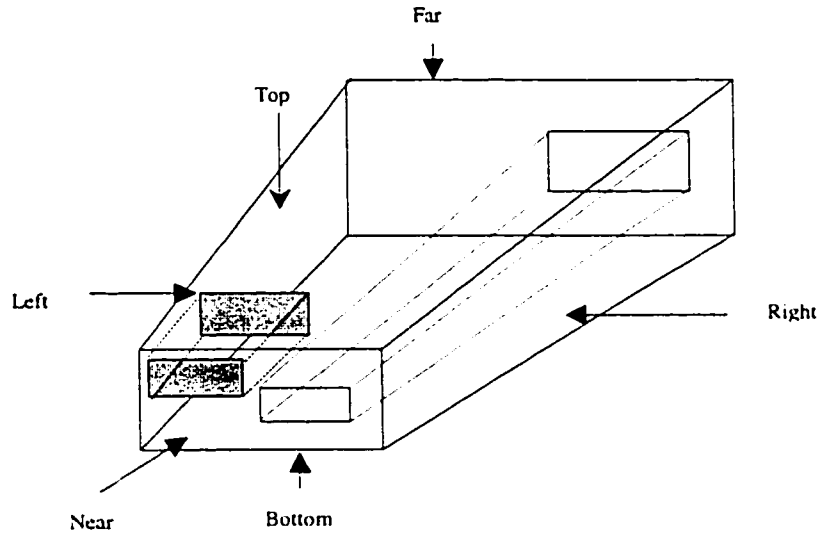

Figure 2-11 The clipping volume for a perspective projection

How can we map 3D coordinates onto a 2D surface? The simplest way is still to ignore one coordinate. Perspective transformations provide a better sense of depth by distant objects with reduced size. The size reduction is a natural consequence of viewing the object from a finite distance. In figure 2-12, E represents the eye of the viewer viewing a screen, the model is behind the screen. An object at P in the model appears on the screen at the point P'. [Pg98]

The point P in the model is $[x, y, z, 1]^T$. The x value does not appear in the diagram becase it is perpendicular to the paper. The transformed coordinates on the screen are (x', y'): there is no z coordinate because the screen has only two dimensions. By similar triangles :

$$x' / d = x / (z+d)$$

$$y' / d = y / (z+d)$$

and hence

$$x' = x*d / (z + d)$$

$$y' = y*d / (z + d)$$

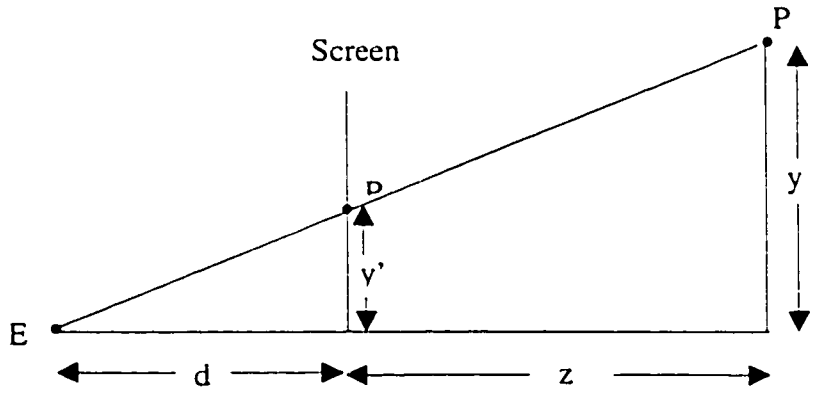

Figure 2-12 Perspective transformation

2.3 3D Physics Concepts

How do we build a 3D simulation of physical object? 3D programs like OpenGL use transformations to render to the screen, but how to we to simulate object's interactions?

2.3.1 Modeling Newtonian Mechanics

To model Newtonian mechanics, the following things are needed:

- Model motion
- Collision detection
- Collision response

2.3.1.1 Modeling Motion

Transformations can be used to model motion, an object's movement could be placed under a transform matrix and its translation modified each frame. In order to calculate these, the parameters such as energy, mass, linear and angular momentum, centre-of-mass, etc. would be needed.

2.3.1.2 Collision Detection

To prevent two objects from occupying the same volume of space, collision detection is needed. Collision detection is quite difficult for arbitrary shapes. Even for rigid objects, there is also considerable overhead when there are a lot of objects to be tested if a collision may happen between every object and any other object.

2.3.1.3 Collision Response

Once a collision happens, the effect of the collision should be calculated, new velocities are generated.

2.3.1.4 Analytical verses Numerical Methods

There are generally two methods to solve the problems outlined above: analytical methods or numerical methods. Analytical methods are used to solve some complex equations, and therefore require a lot of work at design time; numerical methods involve less complex equations but more calculation is done at runtime. In the most general case numerical methods probably are used, but in more restricted cases analytical methods are used.

2.3.2 3D Physics - Solid Objects

Standards such as OpenGL allow the geometry and appearance of solid objects to be defined.

2.3.2.1 Kinematics

In this section, we will discuss some kinematics theory.

2.3.2.1.1 Translation

Translations or relative positions can be represented by vectors of dimension three. In the case of representing a new position relative to the old position, translations can be used mathematically by adding the vectors. If a translation [ax, ay, az] is applied and then another translation [bx, by, bz], the resulting translation will be [ax+bx, ay+by, az+bz]. The usual rules of vector addition are:

[A] + [B] = [B] + [A] = [ax+bx, ay+by, az+bz].

A complex object can be translated by applying the translation to each point on the object. These translations can be used to describe the linear movement of an object.

2.3.2.1.2 Rotation

If an object is not restrained then it can rotate in addition to moving linearly. This gives six degrees of freedom: three for linear movement, three for rotation.

2.3.2.1.3 Uniform Velocity

An object is moving at a constant speed in the x-direction, without rotation, its motion is given by:

$$v = \partial x / \partial t$$

Here, x means moving distance of an object, v means moving speed of an object.

Similarly, an object is moving at a constant speed in the y-direction, without rotation, its motion is given by:

$$v = \partial y / \partial t$$

and an object is moving at a constant speed in the z-direction, without rotation, its motion is given by:

$$v = \partial z / \partial t$$

2.3.2.1.4 Uniform Acceleration

If an object is moving at a constant acceleration, its acceleration is given by:

$$a = \partial v / \partial t$$

Where v means moving speed of an object and t means moving time of an object.

In the more general case of translation in three dimensions:

$$[a] = \begin{bmatrix} \partial^2 x / \partial t^2 \\ \partial^2 y / \partial t^2 \\ \partial^2 z / \partial t^2 \end{bmatrix}$$

Where x, y and z stand for respectively the moving distances of an object in x-direction, y-direction and z-direction.

2.3.2.2 Newtonian Laws

If objects are representing physical objects, we probably calculate the kinematics from the dynamics. For applications such as games and simulations of normal objects we can use Newtonian methods.

Newton defines three laws:

- If no forces act on a particle, the particle retains its linear momentum.
- The rate of change of the linear momentum of a particle is equal to the sum of all forces acting on it.
- When two particles exert forces upon each other, these forces are equal in magnitude and opposite in direction.

2.3.2.3 Motion with No External Torque

What are the equations for the movement of a rigid object with no external forces or torques acting on it? This is the simplest case that we can imagine for physics simulation, so we would like to make sure that we fully understand it and can represent it

programmatically before going on to more complex situations like collisions and jointed structures.

For a rigid object rotating in free space, by Newton's first law.

- Linear momentum of centre-of-mass in x dimension = constant.
- Linear momentum of centre-of-mass in y dimension = constant.
- Linear momentum of centre-of-mass in z dimension = constant.
- Angular momentum about x-axis through centre-of-mass = constant.
- Angular momentum about y-axis through center-of-mass = constant.
- Angular momentum about z-axis through mass-of-mass = constant.

So the linear position of the center of mass is given by:

$$[p] = [p0] + [v]t$$

where:

[p] = position vector in x, y and z dimensions

[p0] = position vector at t=0

[v] = velocity

t = time (scalar)

2.3.2.4 Dynamics Collision

The dynamic equations for a single object were covered in the previous section. Here we extend this analysis to two objects so that we can calculate the result of a collision between the objects.

Linear momentum, angular momentum and energy is passed between the colliding shapes at the point of collision by means of impulse. Impulse is the integral of force over time. This is measured in Newton-seconds. For rigid body collisions, the impact is assumed to happen in an infinitesimally small time.

Collisions in three dimensions are quite a complex problem. If a body is solid and it collides with another body then it may bounce off the other, or slide and/or deform depending on the conditions such as coefficient of friction, elasticity of the bodies, etc. [Web3D]

3. System Design

In this chapter, we introduce several terms that are used in describing the system; define some structures and constants that are used in designing and implementing the project; extract classes that are used in describing 3D objects in the real world; and introduce some algorithms, which are used in describing the relationships between 3D objects, collision detection, as well as collision response, etc.

3.1 Terms

We introduce the following terms, which are used in describing the system.

- Table: A special entity, all other shapes (called entities) can move on it.
- Entity: A specific shape, which would move on a table if a force was applied to it.

3.2 The Definitions of Structure and Constant

For designing and implementing the applications, it is necessary to define some structures, macros, constants and global variables, which would be referenced during the whole processes of design and implementation.

- A macro *ConvertAngle()* is used to convert an angle to a radian. In C++, functions are preferred to macros.

  #define ConvertAngle(*Angle*) ((Angle) * *PI* / 180.0)

  Where *Angle* is a angle, which will be convert to a radian,

  PI is a constant, its value is 3.1416

- A structure *Boundary*

```
typedef struct {
        GLfloat Left;
        GLfloat Right;
        GLfloat Top;
        GLfloat Bottom;
        GLfloat Far;
        GLfloat Near;
} Boundary;
```

is used to describe a boundary of a 3D object. The following algorithm demonstrates how to get boundaries for a cube object:

$Left = X - SL / 2.0$

$Right = X + SL / 2.0$

$Top = Y + SL / 2.0$

$Bottom = Y - SL / 2.0$

$Near = Z + SL / 2.0$

$Far = Z - SL / 2.0$

Here (X, Y, Z) is the coordinate of the cube object, and SL stands for the side length of the cube.

- An enumerated type *DirectionDef* is used to describe an arrow's direction (i.e., a force's direction).

typedef enum {

Left,

Right,

Front,

Back

} ArrowDirections, DirectionDef;

A variable with the value of *Left* means that a force might be applied to a 3D object in the direction from (–X) to (+X); a variable with the value of *Right* means that a force might be applied to a 3D object in the direction from (+X) to (–X); a variable with the value of *Front* means that a force might be applied to a 3D object in the direction from (+Z) to (–Z); and a variable with the value of *Back* means that a force might be applied to a 3D object in the direction from (–Z) to (+Z).

- An enumerated type *EntityPositionStatus* is used to describe a 3D object's position status.

typedef enum {

OnTable,

DropFromLeftEdge,

DropFromFrontEdge,

DropFromRightEdge,

DropFromBackEdge,

FinishDropSimulation

} EntityPositionStatus;

A variable with the value of *OnTable* means that a 3D object (such as a cube) is on the table; a variable with the value of *DropFromLeftEdge* means that a 3D object is dropping down from its left edge of the table; a variable with the value of *DropFromRightEdge* means that a 3D object is dropping down from its right edge of the table; a variable with the value of *DropFromFrontEdge* means that a 3D object is dropping down from its front edge of the table; a variable with the value of *DropFromBackEdge* means that a 3D object is dropping down from its back edge of the table; and a variable with the value of *FinishDropSimulation* means that a 3D object has finished the whole process of dropping down, and has stopped moving. Figure 3-1 illustrates the description of table edges.

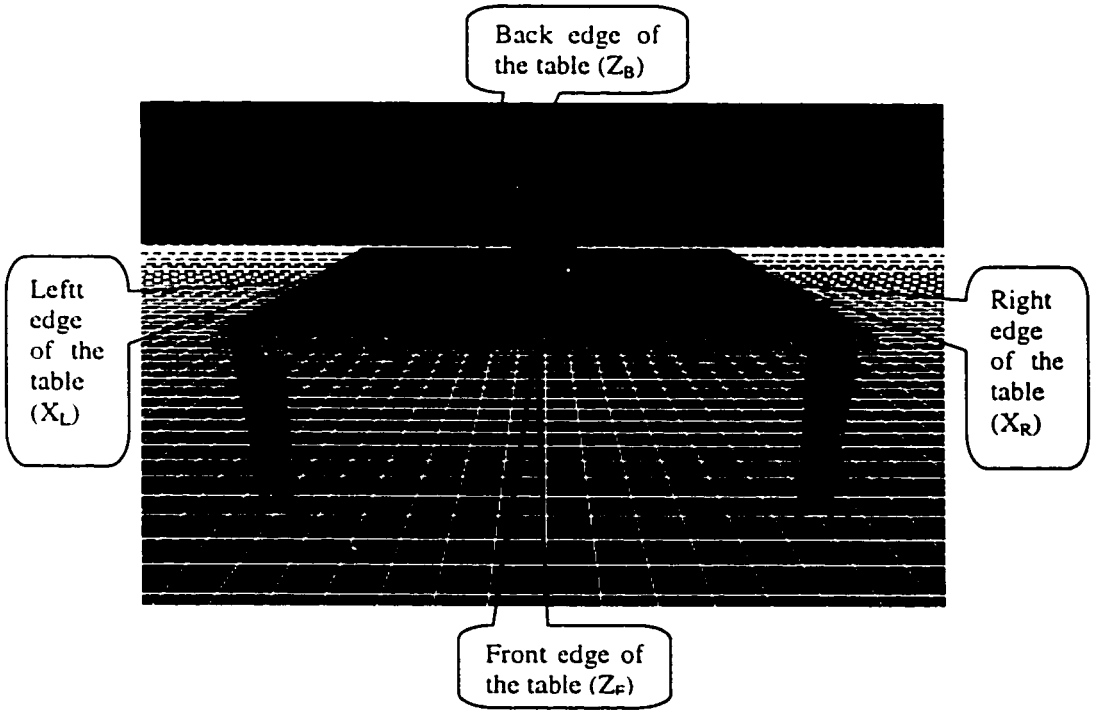

Figure 3-1 Description of table edges

3.3 Class Definitions

In general, a complex 3D object may be considered as combinations of several simple 3D objects, such as cube, sphere and cylinder. Instead of describing a complex 3D object, it may be simplified to describe several simple ones. In particular, since it is very easy to extract common attributes for the simple objects, we define our classes based on this principle.

3.3.1 Class Shapes

Any 3D object has its distinguishing features. For example, a cube has six surfaces (two surfaces are parallel, and they are tangent with another four surfaces), while a sphere has a round shape, the length of its radius always equals to distance between the center to any point of its surface. However, there still are some common attributes for all the simple 3D objects, such as, all shapes have characteristics of color and position.

Shapes is such a class, its member variables are used to describe the common attributes for all simple 3D shapes, and a group of member functions are introduced to maintain the member variables. To make it inheritable, the member functions are defined as public, while member variables are defined as protected.

Actually, *Shapes* is a super class, it describes the common characteristics of simple 3D shape objects. Any specific shape may be described by extending this class *Shapes*.

A vector of *<PositonX, PositionY, PositionZ>* is used to describe the position of a 3D object. The functions

SetPositionX(Glfloat itsPositionX)

SetPositionY(Glfloat itsPositionY)

SetPositionZ(Glfloat itsPositionZ)

are used to set the coordinate values for an object. Once the coordinate values are changed, the object moves. Whenever we want to know where the object is, we should call functions

GetPositionX(), *GetPositionY(),* *GetPositionZ()*

to get the object's position. All the three functions return float numbers, which may be presented as a vector *<X, Y, Z>* to indicate the coordinate value of the object.

A float number *RotationAngle* and a vector *<AxisX, AxisY, AxisZ>* are used to describe object's rotations. Actually, an object may rotate round X-axis, Y-axis, Z-axis or any straight line. When *<AxisX = 1, AxisY = 0, AxisZ = 0>*, the object would rotate round X-axis; when *<AxisX = 0, AxisY = 1, AxisZ = 0>*, the object would rotate round Y-axis; when *<AxisX = 0, AxisY = 0, AxisZ = 1>*, the object would rotate round Z-axis; otherwise, the object would rotate round a straight line, which passes through points <0, 0, 0> and *<AxisX, AxisY, AxisZ>*. Function *SetRotationAngle(GLfloat itsRotationAngle, GLfloat itsX, GLfloat itsY, GLfloat itsZ)* is used to set rotation angle, where *itsRotationAngle* is the angle value. Similarly, the function *GetRotationAngle()* returns the angle value.

SetColor(GLfloat itsColorR, GLfloat itsColorG, GLfloat itsColorB) is used to set display color for a 3D object the parameters are presented by a RGB pair, where *itsColorR* stands for color red, *itsColorG* stands for color green, and *itsColorB* stands for color blue. Their value range are from 0 to 255.

A default constructor is used to initialize the member variable. For example, let the initial rotation angle be zero, rotation axis be Y-axis.

3.3.2 Class Cube

Class *Cube* is derived from class *Shapes*. Obviously, it has some extensions of its super class; the extensions should be able to describe the characteristics of cubes: side length *SideLength* is the most important attribute of cubes. In addition, we introduce a member variable with type of *EntityPositionStatus* (see 3.2 The definitions of structure and constant) to indicate that whether a cube object is on the table, or it is dropping down from table, or the cube has fallen onto the ground.

A pair of functions – *SetSideLength()* and *GetSideLength()* – are used to maintain its member variable *SideLength*. Function *SetSideLength()* assigns a new value to its member variable *SideLength*, while Function *GetSideLength()* returns the value of *SideLength*.

Another member variable *PositionStatus* is maintained by functions *SetPositionStatus()* and *GetPositionStatus()*. *SetPositionStatus()* assigns a specific value to *PostionStatus* --

in the beginning of simulation, the variable *PositionStatus* always has initial value *OnTable*; when the center of the cube object exceeds the edge of table, its member variable *PositionStatus* is assigned by one of the values (*DropFromLeftEdge, DropFromRightEdge, DropFromFrontEdge, DropFromBackEdge*); when the cube object falls to the ground, *FinishDropSimulation* is assigned to the member variable.

For checking if a cube object has reached the edge of the table, a member function *CheckBoundary()* is introduced. It always compares the center position of the object to the table edges, and calls another member function *SetPositionStatus()* to set the member variable *PositionStatus*.

Once the cube object reaches the edge of the table, and is going to drop from the table, a member function *SimulationDropDown()* is called to simulate the whole dropping process. For showing the dropping effects, the whole falling process is divided into several frames, which are shown continuously.

A member function *Show()* simply calls OpenGL API to show a specific cube object on screen.

3.3.3 Class Sphere

Class *Sphere* is derived from class *Shapes*. Obviously, it has some extensions of its super class; the extensions should be able to describe the characteristics of spheres. Radius is the most important attribute of a sphere. In addition, we introduce a member variable

with type of *EntityPositionStatus* (see 3.2 The definitions of structure and constant) to indicate that whether a sphere object is on the table, or it is dropping down from table, or the cube has fallen onto the ground.

A pair of functions – *SetRadius()* and *GetRadius()* are used to maintain its member variable *Radius*. Function *SetRadius()* assigns a new value to *Radius*, while function *GetRadius()* returns the value of *Radius*.

Another member variable *PositionStatus* is maintained by functions *SetPositionStatus()* and *GetPositionStatus()*. *SetPositionStatus()* sets a specific value to *PostionStatus* -- at the beginning of the simulation, the variable *PositionStatus* always has initial value *OnTable*; when the center of the sphere object exceeds the edge of table, its member variable *PositionStatus* is assigned by one of the values (*DropFromLeftEdge*, *DropFromRightEdge*, *DropFromFrontEdge*, *DropFromBackEdge*); when the sphere object has fallen to the ground, *FinishDropSimulation* is assigned to the member variable.

For checking if a sphere object has reached the edge of the table, a member function *CheckBoundary()* is introduced. It always compares the center position of the object to the table edges, and calls another member function *SetPositionStatus()* to set the member variable *PositionStatus*.

Once the sphere object reaches the edge of the table, and is going to drop from the table, a member function *SimulationDropDown()* is called to simulate the dropping process.

A member variable *MovementFactor* is used to object's animations, which is maintained by member functions *SetMovementFactor()* and *GetMovementFactor()*. Whenever a force is applied to a sphere object, a new value is assigned to its member variable *MovementFactor* by calling the member function *SetMovementFactor()*.

For showing the dynamical effects of sphere's movement on table, an idle function *IdleFunction()* is introduced, which always checks the value of member variable *MovementFactor* by calling *GetMovementFactor()*: if *MovementFactor* is greater than zero, the sphere would move a step alone the force's direction, and the member variable *MovementFactor* would be modified by calling function *SetMovementFactor()*.

The idle function is also used to simulate the dropping process of a sphere object. When a sphere object has reached an edge of the table and begins dropping, the coordinate of the sphere would be changed continuously, the idle function might show the sphere dynamically according to the different coordinates.

A member function *Show()* simply calls OpenGL API to show a specific sphere object on screen.

3.3.4 Class Cylinder

Class *Cylinder* is derived from class *Shapes*. Obviously, it has some extensions of its super class; the extensions should be able to describe the characteristics of cylinder. Base

radius, top radius and cylinder height are the most important attributes of a cylinder. In this project, the cylinder object is used only to draw the legs of a table.

A group of member variables and member functions are introduced to describe cylinder objects. Function *SetBaseRadius()* is used to assign a new value to its member variable *BaseRadius*, while function *GetBaseRadius()* returns the value of *BaseRadius*; function *SetTopRadius()* is used to assign a new value to its member variable *TopRadius*, while function *GetTopRadius()* returns the value of *TopRadius*; function *SetCylinderHeight()* is used to assign a new value to its member variable *CylinderHeight*, while function *GetCylinderHeight()* returns the value of *CylinderHeight*.

A member function *Show()* simply calls OpenGL API to show a specific cylinder object on screen.

3.3.5 Class SolidBox

Class *SolidBox* is derived from class *Shapes*. Obviously, it has some extensions of its super class; the extensions should be able to describe the characteristics of solid box. Width, height and depth are the most important attributes of a solid box. In this project, the solid box object is used only to draw the surface of a table.

A group of member variables and member functions are introduced to describe solid box objects. Function *SetWidth()* is used to assign a new value to its member variable *Width*, while function *GetWidth()* returns the value of *Width*; function *SetHeight()* is used to

assign a new value to its member variable *Height*, while function *GetHeight()* returns the value of *Height*; function *SetDepth()* is used to assign a new value to its member variable *Depth*, while function *GetDepth()* returns the value of *Depth*. A member function *Show()* simply calls OpenGL API to show a specific solid box object on screen.

3.3.6 Class Arrow

Class *Arrow* is derived from class *Shapes*. Several member variables and functions are used to maintain an arrow object.

A member function *Show()* draws an arrow to indicate in which direction a force would be added to a 3D object. The arrow object would move together with the object that a force acted on. If a force direction changes, the arrow changes its direction. We suppose that a force is always vertical to one of the object surfaces.

3.4 Position Relationship Between Entities and Table

When an entity is created, it should be on a table. Once a force is added to the entity, it should move on the table. When the entity reaches the edge of the table, and the force is still applied, the entity drops down from the table. Therefore we say that the boundaries of the table are the critical status of an entity (see figure 3-1 description of table edges).

Let coordinate value of an entity be (X, Y, Z), when the position of the entity satisfies the one of following conditions:

$X > X_R$, *or*

$X < X_L$, *or*

$Z > Z_F$, *or*

$Z < Z_B$

The entity might drop down from the table, as shown in figure 3-2.

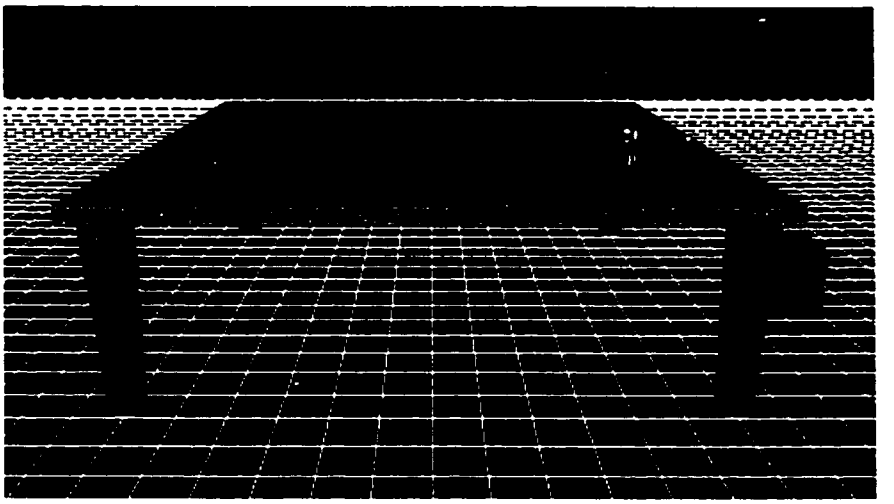

Figure 3-2 Entity drop down from table

As an example, in this project we describe the position relationships between two cubes.

- Both two cubes are on a table. In this case when a cube moves, it may collide another one, as shown in figure 3-3.

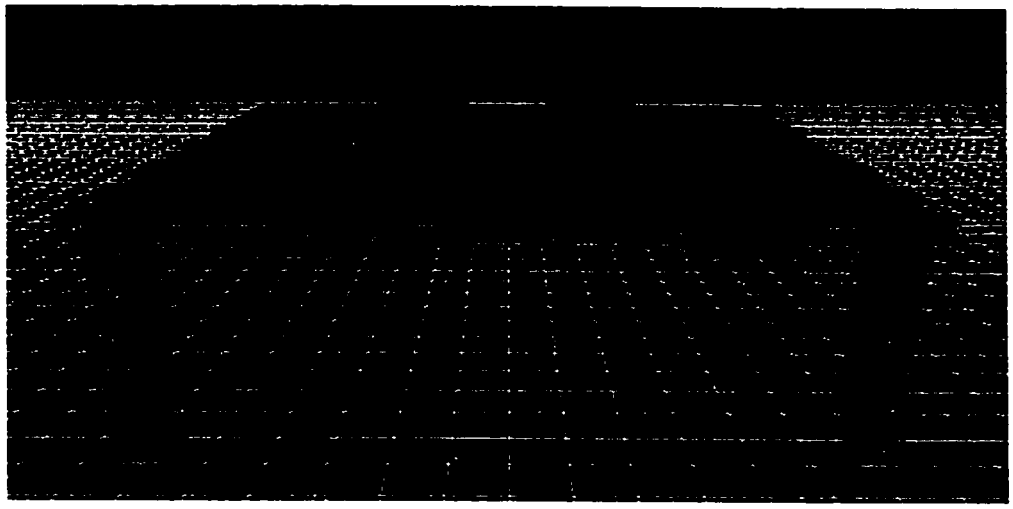

Figure 3-3 Both two cubes are on a table

- A cube is on another cube. If the small cube is pushed, it would move on the big cube, finally it may drop down from the big one. However, if the small cube is on the big cube, and a force is added to the big cube, both cubes would move together, as shown in figure 3-4.

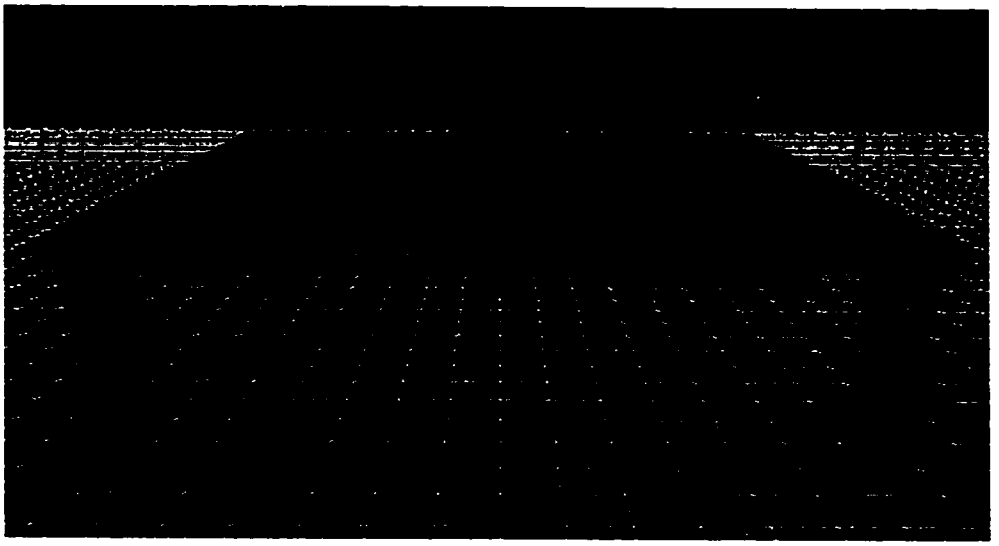

Figure 3-4 A cube is on another cube

3.5 Assumptions and Principles

First of all, we make some assumptions to simplify our discussion.

- **Force and its action point**

  An entity would move only when a force is applied to it. To apply a force to an entity, the force direction and action point should be described. With a cube, we assume that a force always acts on a center point of a surface, and is tangent with the face; with a sphere, we assume that a force is always parallel with the X-axis (or Y-axis) of the coordinate system. The figures 3-5 and figure 3-6 demonstrate the cases (the arrows present the directions and action points of forces).

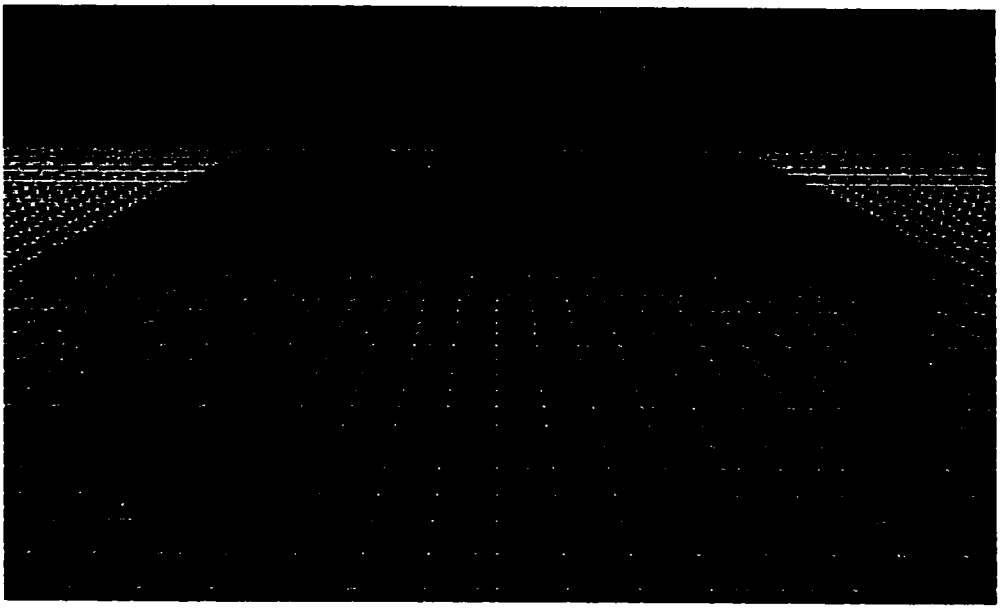

Figure 3-5 A force acts on a cube

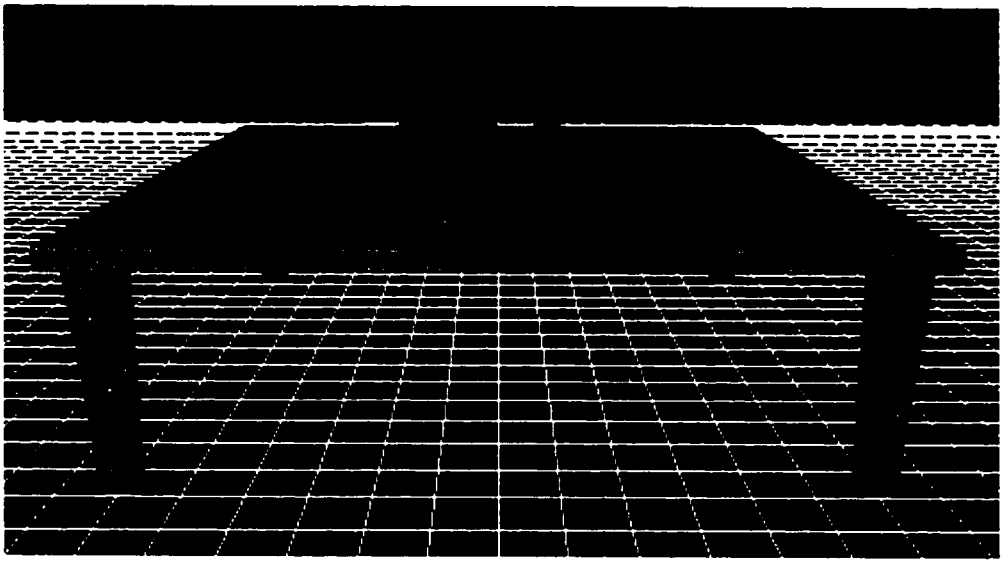

Figure 3-6 A force acts on a sphere

- **What would happen when a force acts on an entity**

  When a force is applied to an entity, the entity would move a distance along the force's direction. However, before an entity moves, a collision detection must be done to avoid multi-entities occupy the same volume of space.

  Let *CubeA* stands for the object of cube A, *CubeA*'s side length is *LengthA*; *CubeB* stands for the object of cube B, *CubeB*'s side length is *LengthB*. We say *CubeA* and *CubeB* are not alignment, if

  (CubeB.X - CubeB.LengthB/2.0 >= CubeA.X - CubeA.LengthA/2.0) &&

  (CubeB.X + CubeB.LengthB/2.0 <= CubeA.X + CubeA.LengthA/2.0)

  or

  (CubeA.X - CubeA.LengthA/2.0 >= CubeB.X - CubeB.LengthB/2.0) &&

  (CubeA.X + CubeA.LengthA/2.0 <= CubeB.X + CubeB.LengthB/2.0)

  or

  (CubeA.Z - CubeA.LengthA/2.0 >= CubeB.Z - CubeB.LengthB/2.0) &&

  (CubeA.Z + CubeA.LengthA/2.0 <= CubeB.Z + CubeB.LengthB/2.0)

  or

  (CubeB.Z - CubeB.LengthB/2.0 >= CubeA.Z - CubeA.LengthA/2.0) &&

  (CubeB.Z + CubeB.LengthB/2.0 <= CubeA.Z + CubeA.LengthA/2.0)

  where *CubeA.X* stands for *CubeA*'s x-coordinate,

  CubeA.Z stands for *CubeA*'s z-coordinate,

  CubeB.X stands for *CubeB*'s x coordinate,

  CubeB.Z stands for *CubeB*'s z coordinate,

CubeA.LengthA stands for *CubeA*'s side length,

CubeB.LengthB stands for *CubeB*'s side length,

"*&&*" stands for LOGIC AND.

How do we consider the case that two entities collide, but they are not alignment (as shown in figure 3-7)?

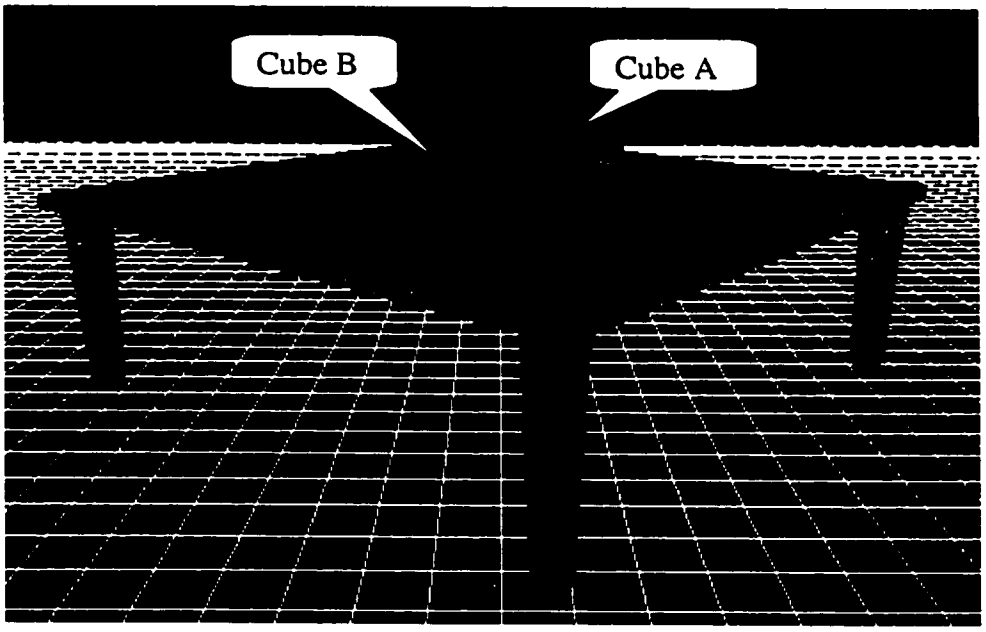

Figure 3-7 Two cubes are not alignment completely

If this happens, each entity should move along different curve trajectory. However, it is very difficult to calculate the movement trajectories. We simplify this into such a simple case: when a force acts on the cube A, it would be transmitted to the center of cube B in the same direction. Thus, both the entities would move along tracks of straight lines, not curves.

3.6 Collision Detection

To prevent two objects from occupying the same volume of space, we need collision detection. Collision detection is quite difficult. This might also be a heavy overhead – N^2 for N objects, if there are a lot of objects. In this project, we only discuss the simple case.

3.6.1 Collision Detection of Two Cubes

Suppose that there are two cubes on a table – *Cube1* and *Cube2*. *Cube1* has a position of (*X1*, *Y1*, *Z1*) with side length of *SideLength1*, and *Cube2* has a position of (*X2*, *Y2*, *Z2*) with side length of *SideLength2*. Let rotation angles surrounding Y-axis be zero, if a force is applied in X-direction (or –X direction), a collision would happen only when following condition is true:

```
(( abs( X1 – X2 ) < SideLength1 / 2.0 + SideLength2 / 2.0 ) && ((Z2 – SideLength2 / 2.0 <= Z1 –
SideLength1 /2.0 <= Z2 + SideLength2 / 2.0) || (Z2 – SideLength2 / 2.0 <= Z1 + SideLength1
/2.0 <= Z2 + SideLength2 / 2.0)))
||
(( abs( X1 – X2 ) < SideLength1 / 2.0 + SideLength2 /2.0 ) &&((Z1 – SideLength1 / 2.0 <= Z2 –
SideLength2 / 2.0 <= Z1 + SideLength1 / 2.0) || (Z1 – SideLength1 / 2.0 <= Z2 + SideLength2 /
2.0 <= Z1 + SideLength1 / 2.0)))
```

If a force is applied in Z-direction (or –Z direction), a collision would happen only when following condition is true:

((abs(Z1 – Z2) < SideLength1 / 2.0 + SideLength2 / 2.0) && ((X2 – SideLength2 / 2.0 <= X1 – SideLength1 <= X2 + SideLength2 / 2.0) || (X2 – SideLength2 / 2.0 <= X1 + SideLength1 <= X2 + SideLength2 / 2.0)))

||

((abs(Z1 – Z2) < SideLength1 + SideLength2) && ((X1 – SideLength1 <= X2 – SideLength2 <= X1 + SideLength1) || (X1 – SideLength1 <= X2 + SideLength2 <= X1 + SideLength1)))

where *abs* stands for absolute value of a number,

"&&" stands for logic and,

"||" stands for logic or,

sqrt stands for square root of a number

Now we explain and justify the collision conditions -- we only verify one case here, others are same. Consider the case of two cubes, where a force is applied in X-direction. If *abs(X1 – X2) < SideLength1 / 2.0 + SideLength2 / 2.0*, that means the two cubes overlap in X-direction; at this moment, if *Z2 – SideLength2 / 2.0 <= Z1 – SideLength1 / 2.0 <= Z2 + SideLength2 / 2.0*, or *X2 – SideLength2 / 2.0 <= X1 + SideLength1 <= X2 + SideLength2 / 2.0*, that means the two cubes overlap in Z-direction. When two cubes overlay in both X-direction and Z-direction, a collision happens.

3.6.2 Collision Detection of Two Spheres

Suppose that there are two spheres on a table – *Sphere1* and *Sphere2*. *Sphere1* has a position of (*X1, Y1, Z1*) with a radius of *SphereRadius1*, while *Sphere2* has a position of

(*X2*, *Y2*, *Z2*) with a radius of *SphereRadius2*. A collision happens only when the following condition is true:

sqrt(power(X1 – X2) + power(Y1 – Y2) + power(Z1 – Z2)) <= SphereRadius1 + SphereRadius2

where *sqrt* stands for square root of a number,

power stands for square of a number.

3.7 Collision Response

Once we have detected the collision, we need to calculate the effect of the collision, by generating new coordinate positions for each entity.

- **Two cubes collide**

  When two cubes collide, each cube would move same distance on the table along force direction, if cubes will not drop down (i.e., the target positions are still inside the table), otherwise a cube would drop down to the ground.

- **Two sphere collide**

  When two spheres collide, for example, a force is added to the small sphere, and transmitted to the big one, as shown in figure 3-8.

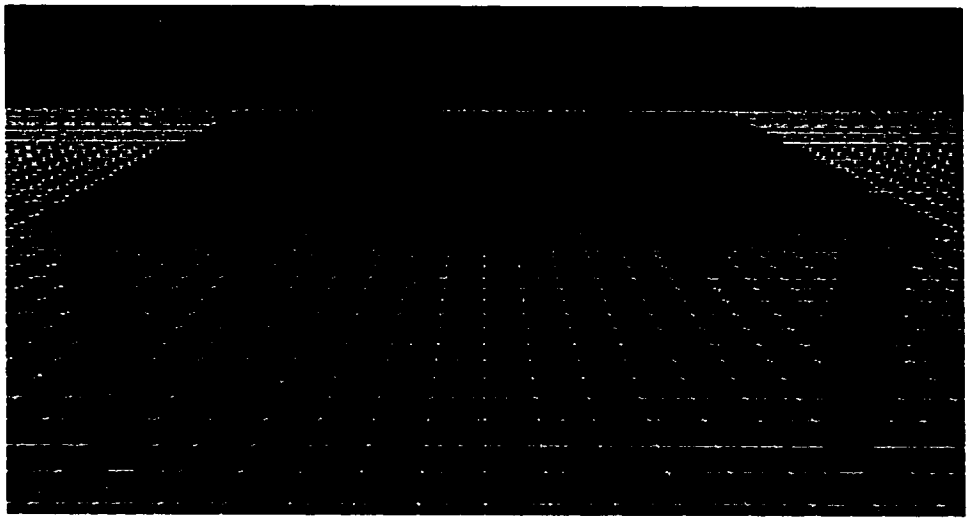

Figure 3-8 A force is added and transmitted

the big one would roll a distance -- *Step1*; while the small one would roll a distance -- *Step2*, where *Step1* >= *Step2*.

However, during the spheres moving, we should check whether the spheres would drop down from the table. In general, if the following conditions

$X_L <= Sphere.PositionX + itsStep <= X_R$, *and*

$Z_B <= Sphere.PositionZ + itsStep <= Z_T$

Where *Sphere.PositionX* stands for the sphere's current X coordinate; *Sphere.PositionZ* stands for the sphere's current Z coordinate; *itsStep* stands for a distance that the sphere would scroll; X_L stands for the left boundary of the table; X_R stands for the right boundary of the table; Z_B stands for the back boundary of the table; Z_T stands for the front boundary of the table.

are satisfied, the sphere would still stay on the table, otherwise, the sphere would drop down from the table.

3.8 Response When A Force Is Applied to An Entity

Whenever a force is added to an entity, the entity would move, and the new position where the entity will move could be calculated by general algorithm. Before an entity moves collision detection must be performed. How do entities move when a force is applied?

- **Single entity on a table**

  In the case of a single entity, we don't need to detect any collision. The only thing we should check is whether an entity has reached a boundary of the table, if it has, a drop simulation will be issued.

  Let *Angle* be the angle between a force and the X-axis, as shown in figure 3-9.

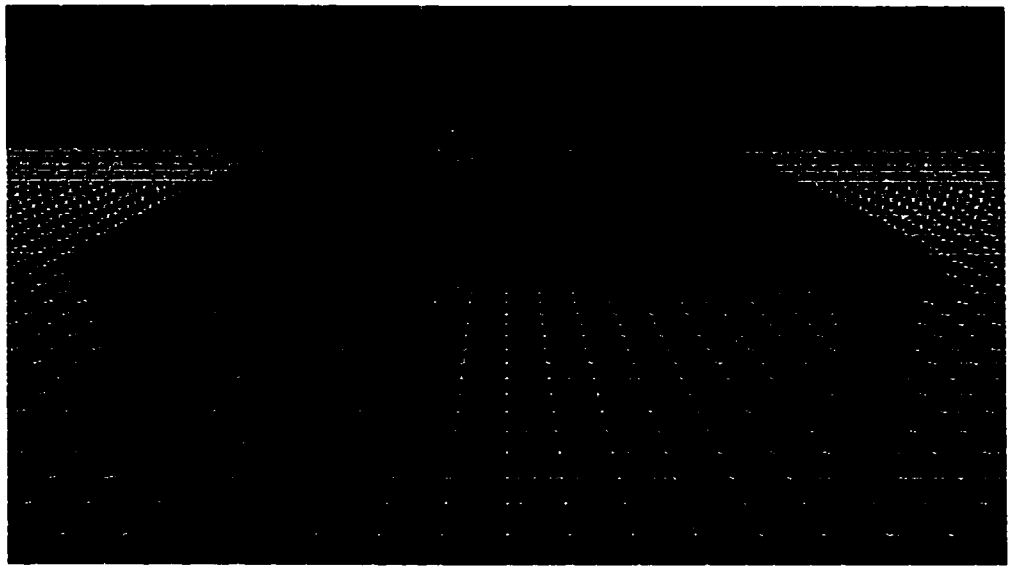

Figure 3-9 Single entity on a table

Then the target position that the entity will move to can be calculated by:

$$PositionX = CurrentPositionX + Step * cos(Angle)$$

$$PositionY = CurrentPostionY$$

$$PositionZ = CurrentPositionZ - Step * sin(Angle)$$

If the target position of the cube exceeds the boundary of the table, the entity would move to the boundary first, and then drop down from the table.

- **Two cubes on a table**

  In the beginning of simulation, two cubes are apart from each other, as shown in figure 3-10.

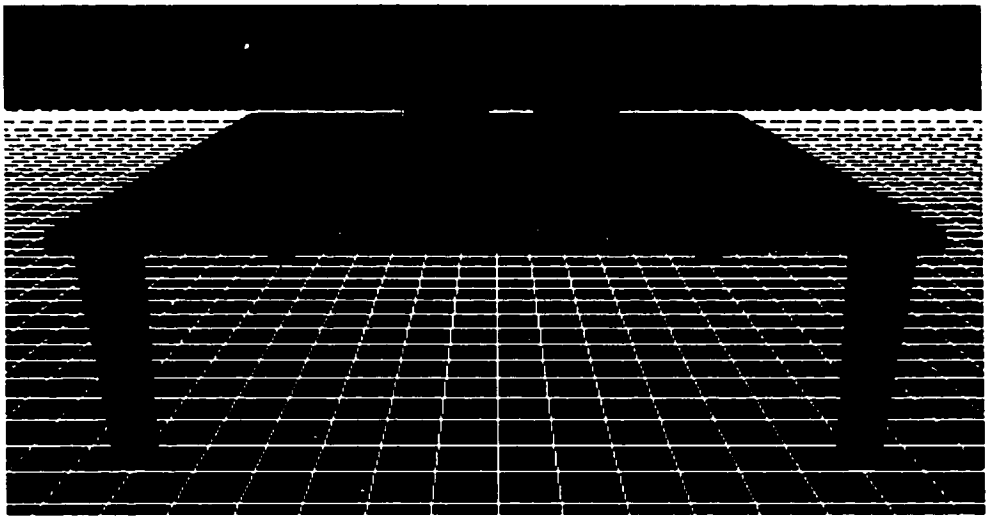

Figure 3-10 Initial positions of two cubes

When a force is applied to the left cube, it would move a distance along the force's direction, and the right one keeps unchanged. If we keep pushing the left cube, a collision would happen, as shown in figure 3-11.

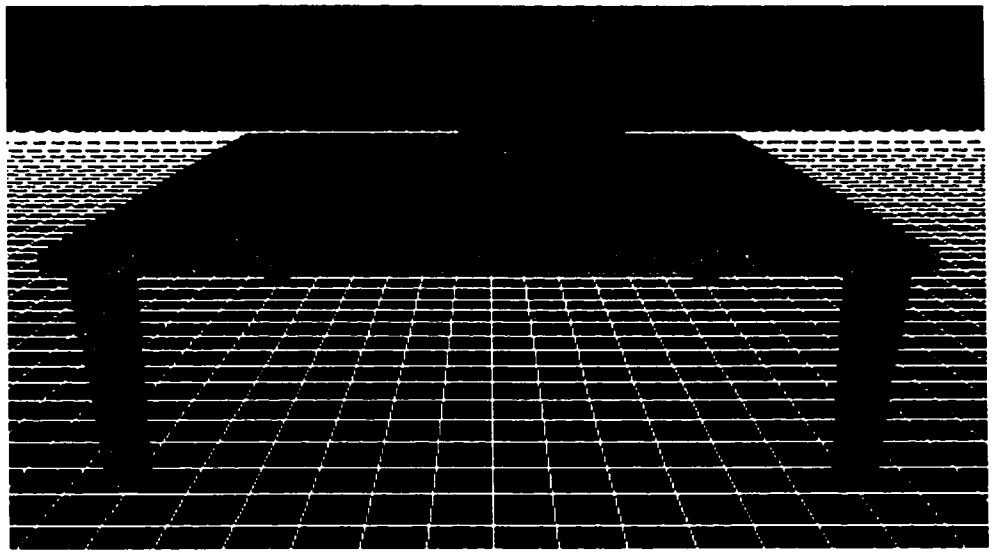

Figure 3-11 Collision happens

In this case, if we continue to the left cube in same direction, both the cubes would move together along in the same direction. When the right cube reaches the boundary of table, as shown in figure 3-12.

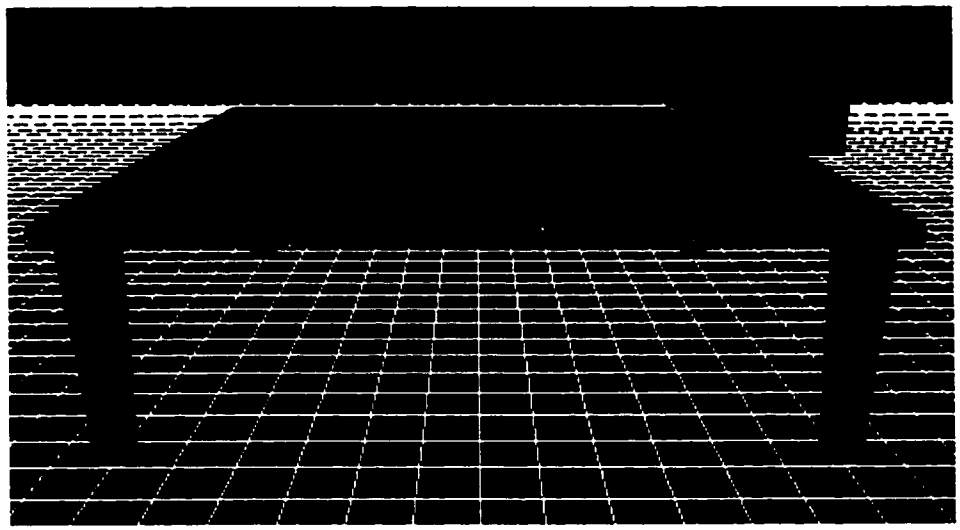

Figure 3-12 A cube reaches the boundary of the table

it would drop down from the table, as shown in 3-13.

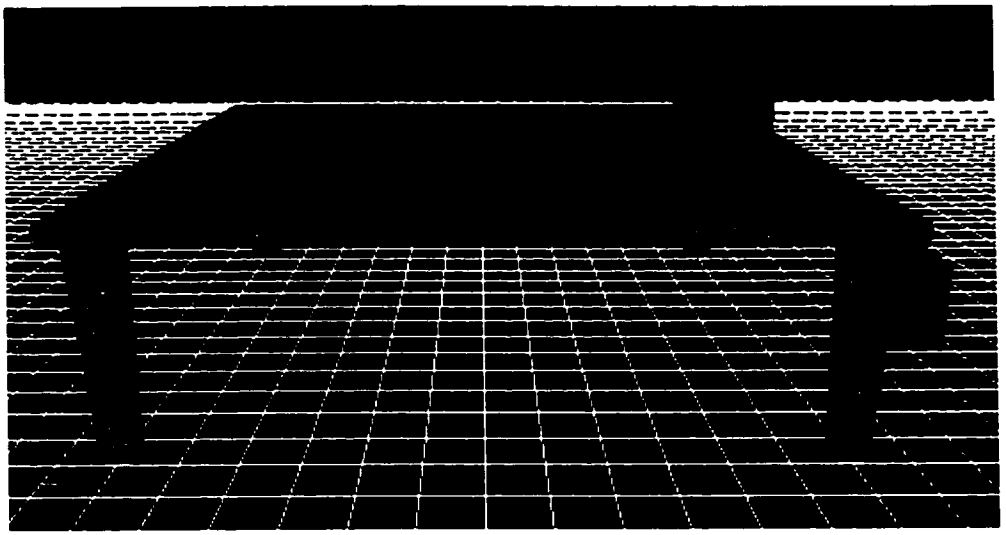

Figure 3-13 A cube drops down from the table

- **Two cubes, one is on top of another**

  In the beginning of the simulation, we suppose a small cube is on top of a big one, as shown in figure 3-14.

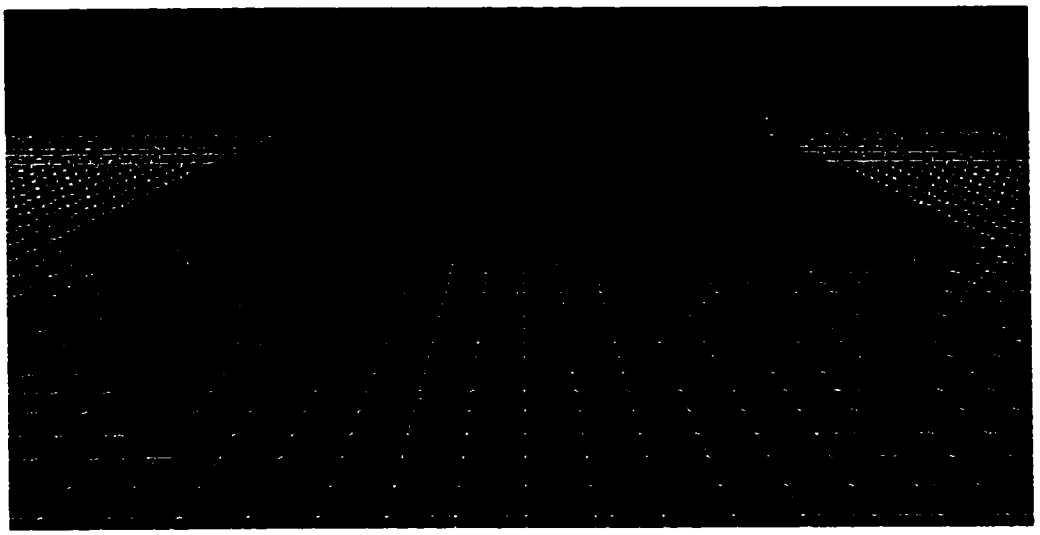

Figure 3-14 A small cube is on the big cube

When a force is applied to the big cube, both cubes would keep moving together like one entity (as shown in figure 3-15). Once the big cube reaches the boundary of the table (as shown in figure 3-16), and the force is still applied, both the cubes would drop down together, as shown in figure 3-17.

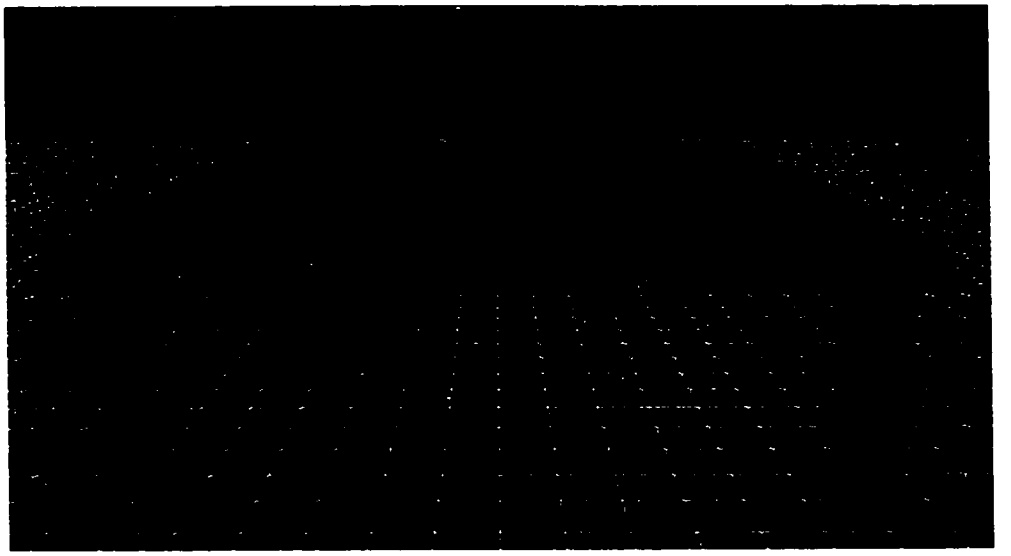

Figure 3-15 Two cubes move together

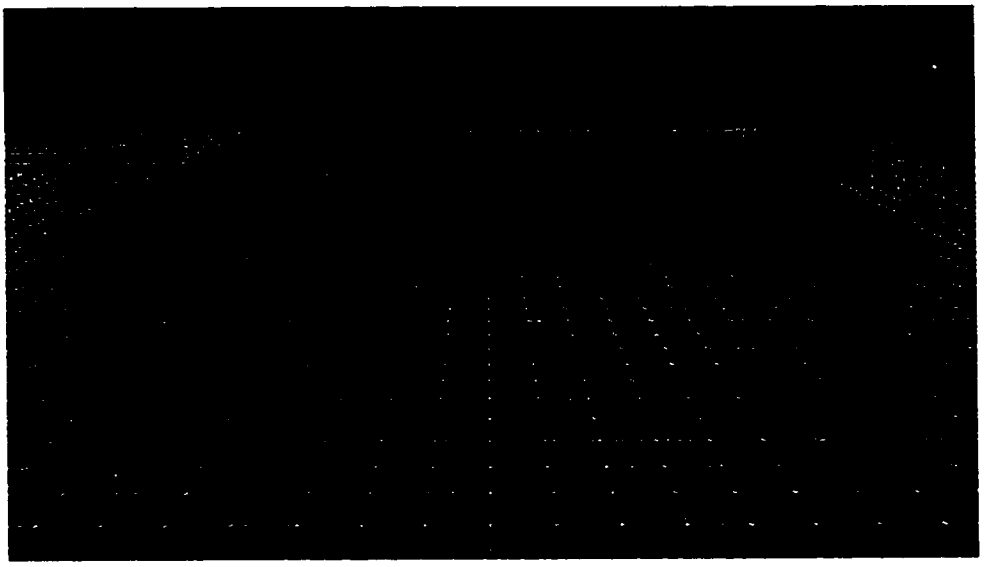

Figure 3-16 One cube reaches the boundary of the table

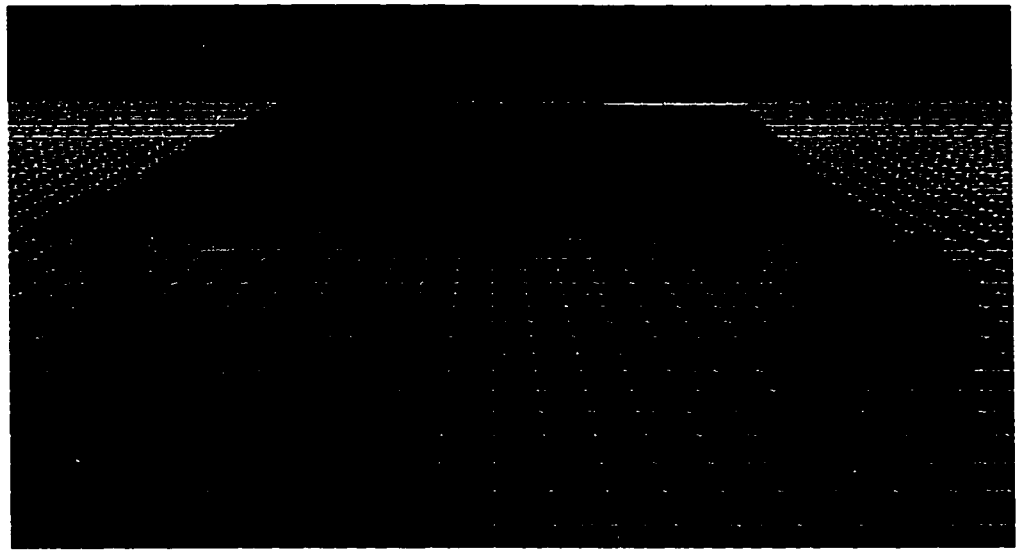

Figure 3-17 Two cubes drop down together

When a force is applied to the small cube, as shown in figure 3-18,

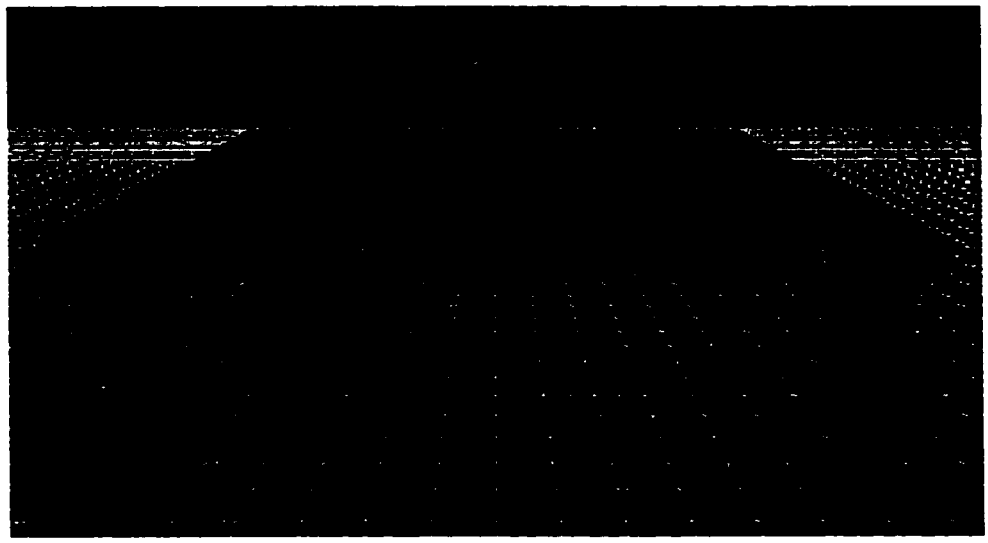

Figure 3-18 A force is applied on the small cube

it would move on the big cube until it reaches the boundary of the big cube, and falls to the table, as shown in figure 3-19 and figure 3-20.

Figure 3-19 A small cube reaches the boundary of the big cube

Figure 3-20 A small cube drops down to the table

From this on, the current status is exactly same as "two cubes on a table".

4. Simulation Results

In this chapter, we use four scenarios to show 3D objects worlds. Each simulation is completed based on some initial conditions. A real 3D object simulation should not only be concerned with geometry properties, but should also appear to model physical properties, such as mass, center of gravity, velocity, acceleration, etc. This means that behaviors will follow directly from these properties. In our project, we start from object's geometry properties; other physical properties will be added to framework in the future.

4.1 Example One

The first example presents a simulation of a single cube object. It will show a series of activities including object rotation, movement, zoom in, zoom out, and dropping from table.

The initial positions are shown in Figure 4-1.

Figure 4-1. Initial position of objects

Step 1. When you start the program

When you start the program, you will see two objects on the screen: a cube and a table. The cube is positioned at the center of the table.

Step 2. Set the position and direction of a force

Click on menu 'Push Direction', an arrow will appear on the cube; press Space key to adjust arrow direction and position.

Step 3. Start simulation

- Select 'Table and Entities' from menu 'Set Rotation Entities', the cube and table will rotate around the center point of table, the rotation angle is depended on the variable Table :: itsRotationAngle. See figure 4-2.
- Select menu 'Entities on the table" from 'Set Rotation Entities', the cube will rotate around the center point of cube; the rotation angle depends on the variable Cube :: itsRotationAngle. See figure 4-3.
- Select 'Zoom In' from zoom menu, objects will be magnified; 'Zoom Out' will minify the objects.
- Select 'Rotate clockwise' or 'Rotate counterclockwise' from 'Rotation' menu, we can rotate objects in clockwise or counterclockwise direction.
- Select 'Reset', objects would return to initial positions.
- Select 'Move entity', the cube will move a distance toward the arrow direction. See figure 4-4.
- Whenever center point of cube passed over the border of table, the cube will drop down from the table. See figure 4-5.

Figure 4-2 Cube and table rotate around the center point of table

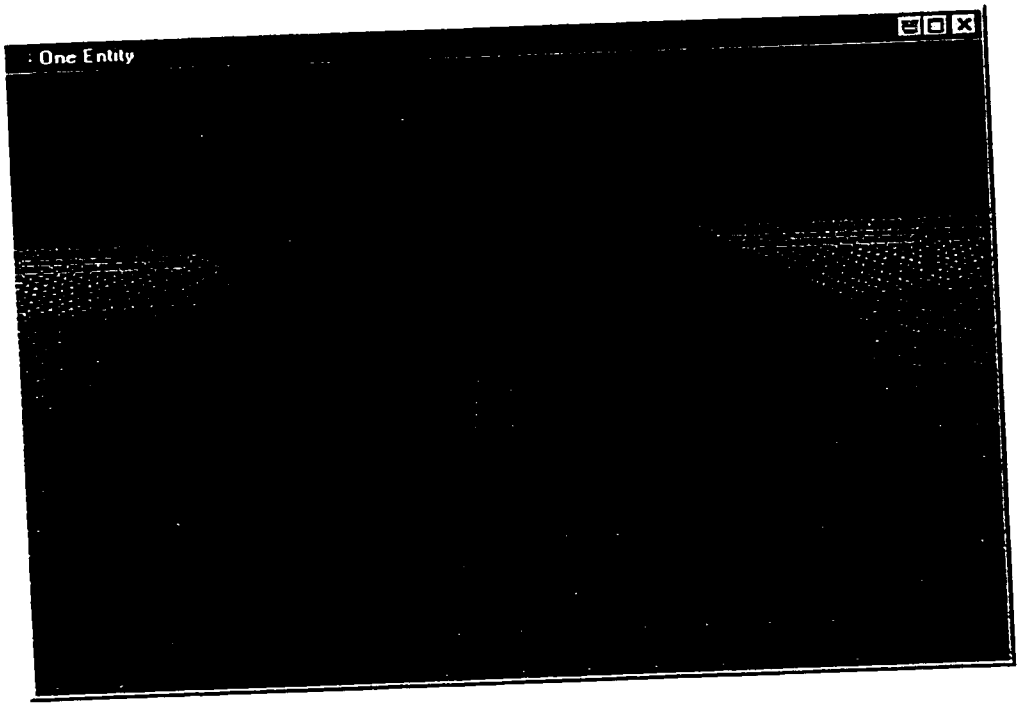

Figure 4-3 Cube rotates around its center

Figure 4-4 Cube moves a distance toward the arrow direction

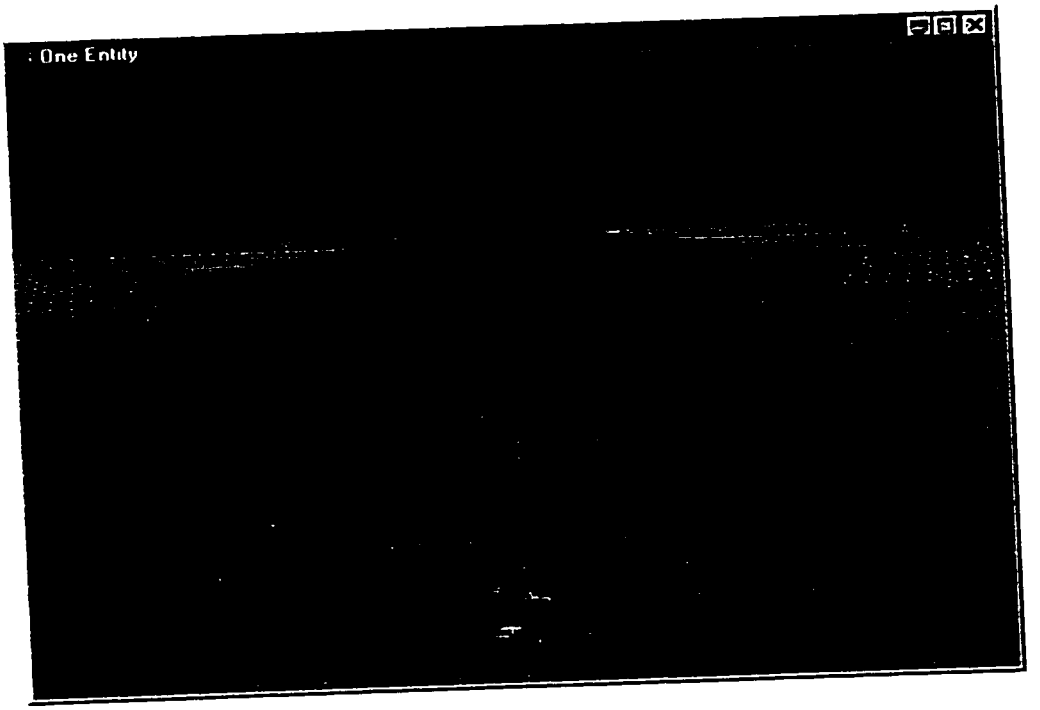

Figure 4-5 Cube drops down from the table

4.2 Example Two

This example will present a simulation of two cube objects. It will show a series of activities including object rotation, movement, zoom in, zoom out, and drop from table. The initial positions are shown in table 4-6.

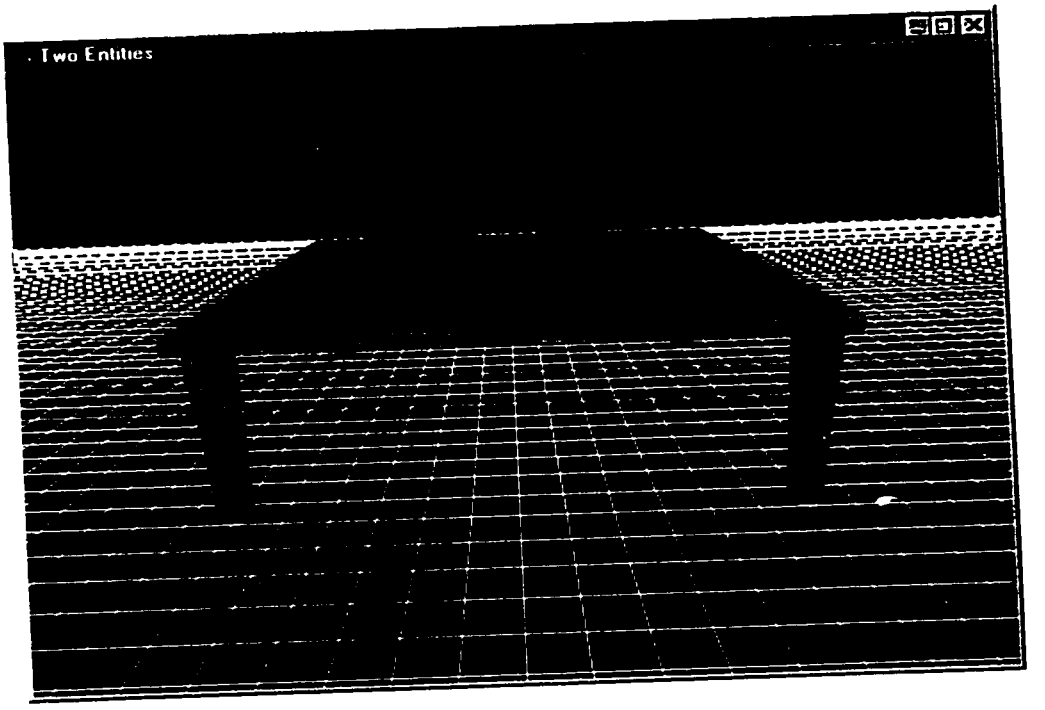

Figure 4-6 Initial positions

Step 1. Start the program

The user interface starts as shown in figure 4-6. Before simulation starts, it displays three objects: two cubes and a table. The cubes are put on the table.

Step 2. Set position and direction of a force

Select 'On' from the menu 'Push Direction", an arrow will appear onto a cube; press Space key to adjust arrow direction and position; select 'Off' from the menu 'Push Direction", the arrow will disappear.

Step 3. Start simulation

- Select menu 'Rotation', the cubes and table will rotate a specified angle around the center of table. See figure 4-7.

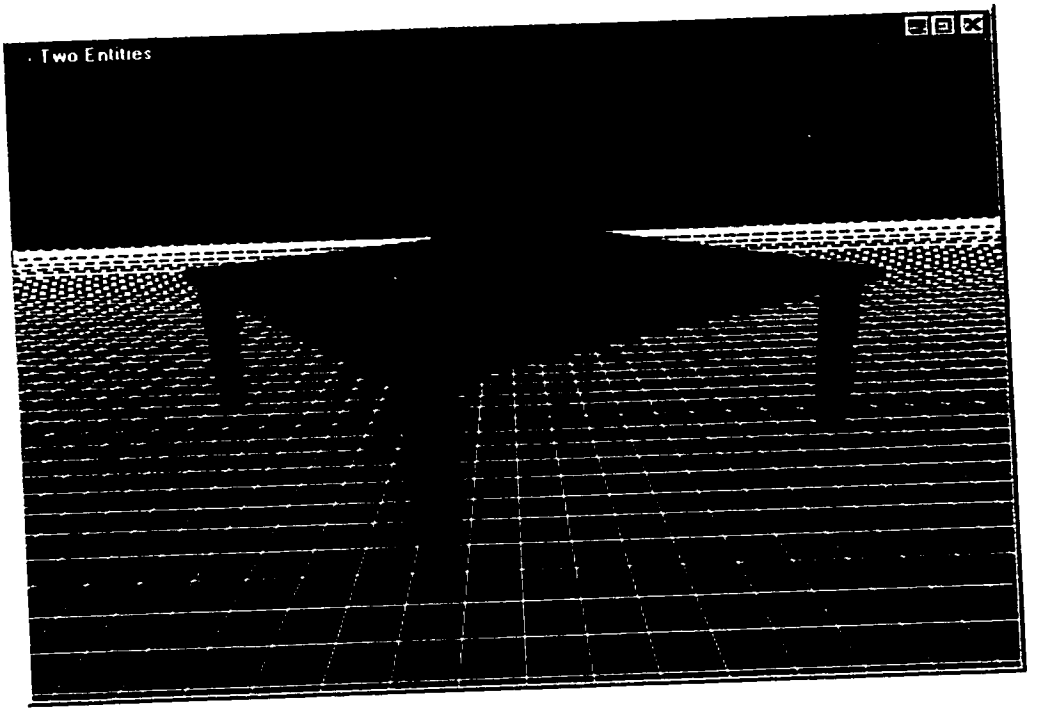

Figure 4-7 Table rotates a specified angle around the center of table

- Select menu 'Zoom In / Zoom Out', objects will be magnified or reduced. See figure 4-8.
- Press '+' or '-' key, objects will rotate a specified angle.
- Click 'Reset', objects return to initial positions.
- Press 'G' key, the cube with force acting on it will move a distance toward the arrow direction. When it meets another cube, both of them will move together in the direction of the arrow. See figure 4-9.
- Whenever the center point of a cube passes over the border of the table, the cube will drop down from the table. See figure 4-10.

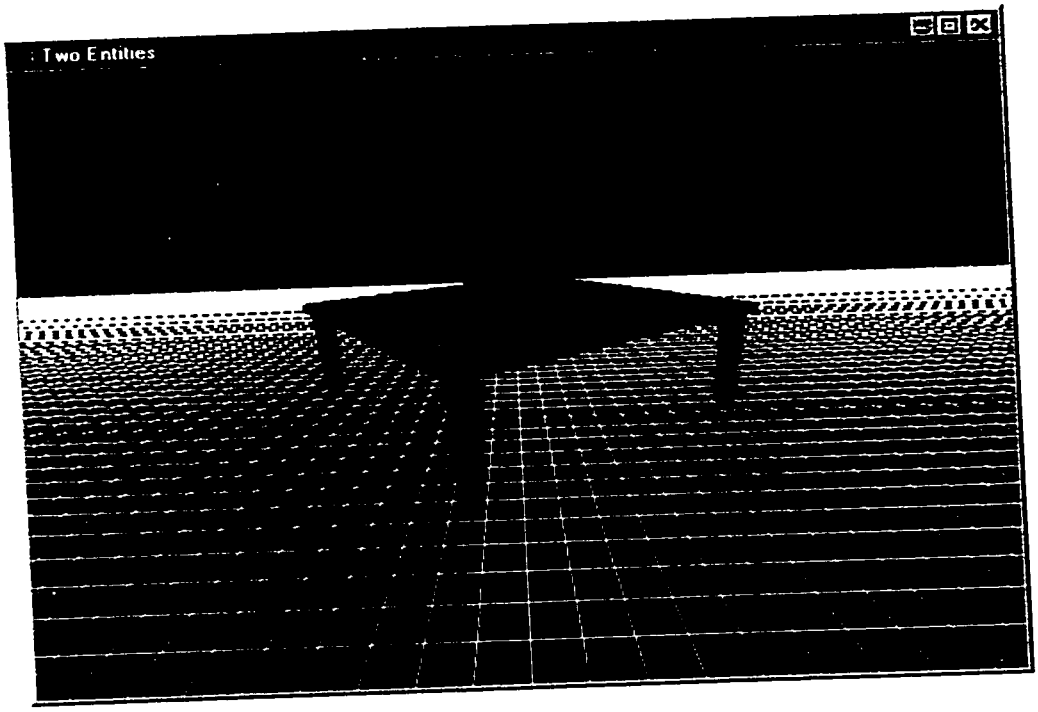

Figure 4-8 Objects are minified

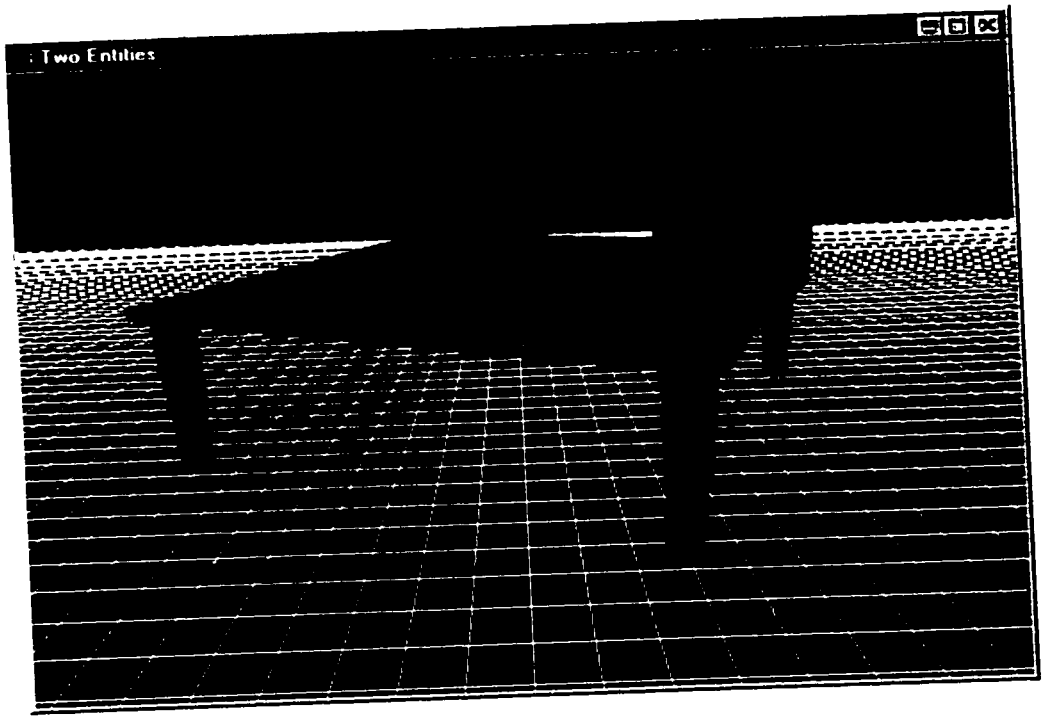

Figure 4-9 Two cubes move together toward the arrow direction

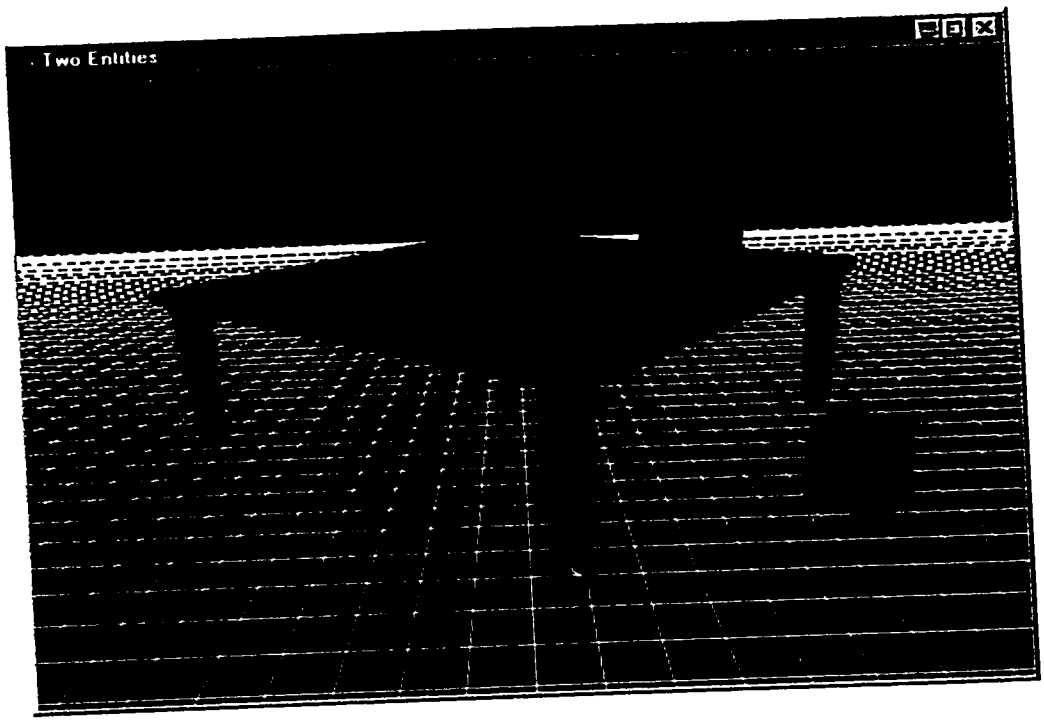

Figure 4-10 A cube will drop down from the table

4.3 Example Three

This example will present a simulation of two cube objects (one is on another one). It will show a series of activities, including object rotation, movement, zoom in, zoom out, and drop from table.

The initial positions are shown in table 4-11.

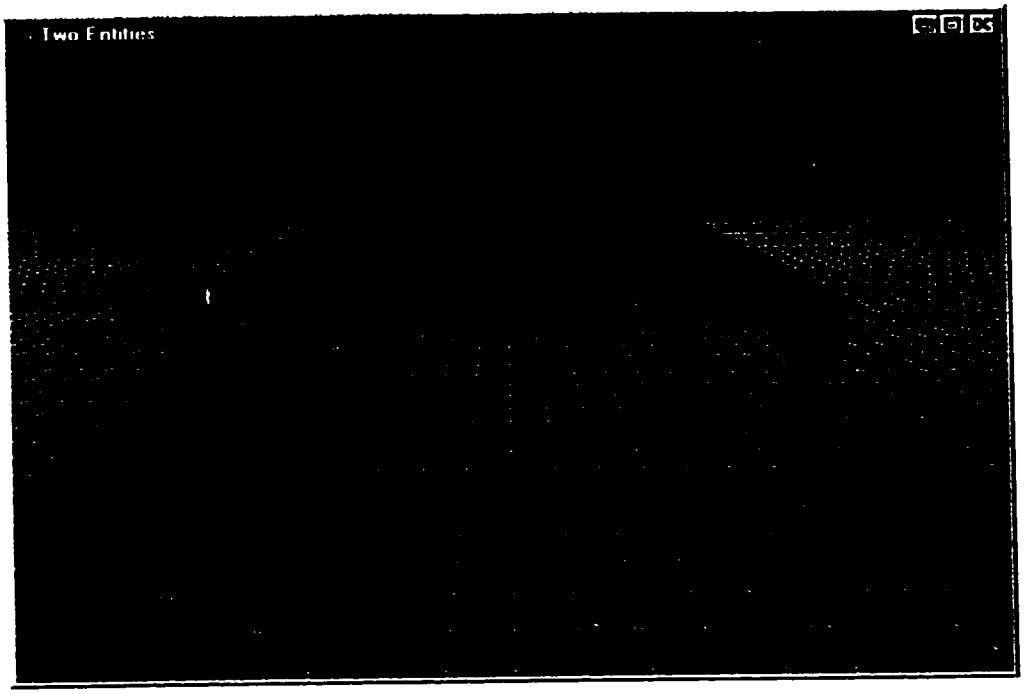

Figure 4-11 Initial positions

Step 1. Start the program:

The user interface starts as shown in figure 4-11. Before simulation, one cube is on the top of another one.

Step 2. Set position and direction of a power

Select menu 'Push Direction', an arrow will appear onto the cube; press Space key to adjust the arrow direction and position.

Step 3. Start simulation

- Select menu 'Rotation', the cubes and table will rotate an angle. See figure 4-12.
- Select menu 'Zoom In / Zoom Out", objects will enlarge or diminish. See figure 4-13.
- Select 'Reset' menu, objects will be reset to their initial positions.
- Press 'G' key, the cube that is pushed will move a step toward the arrow direction. If the upper one reaches the border of another one, it will drop down to the table or ground. See figure 4-14.

Figure 4-12 Cubes and table rotate an angle

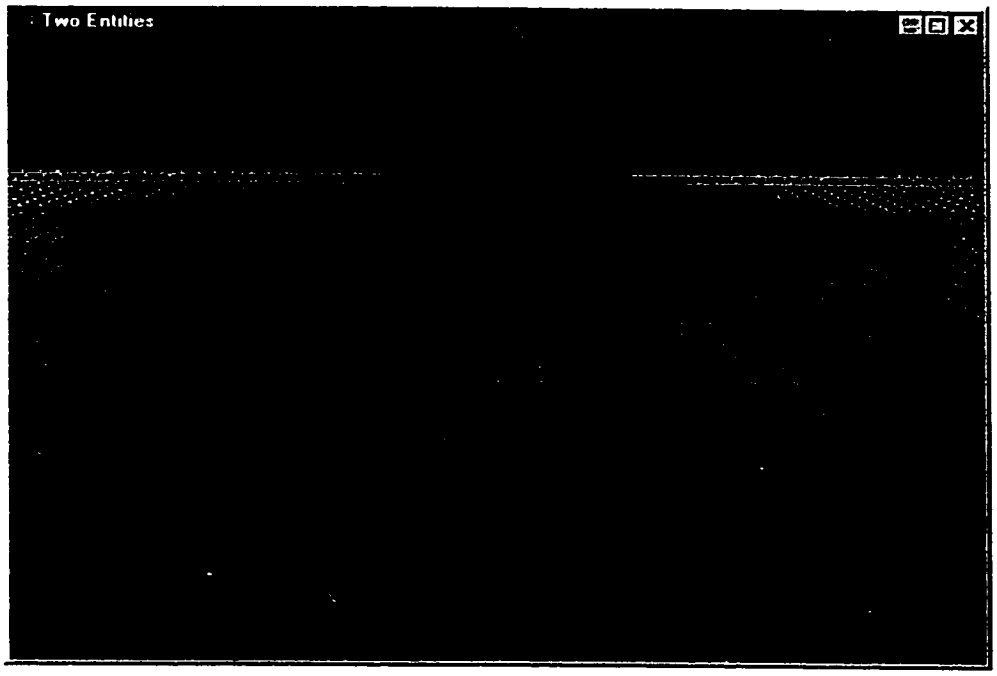

Figure 4-13 Objects enlarge

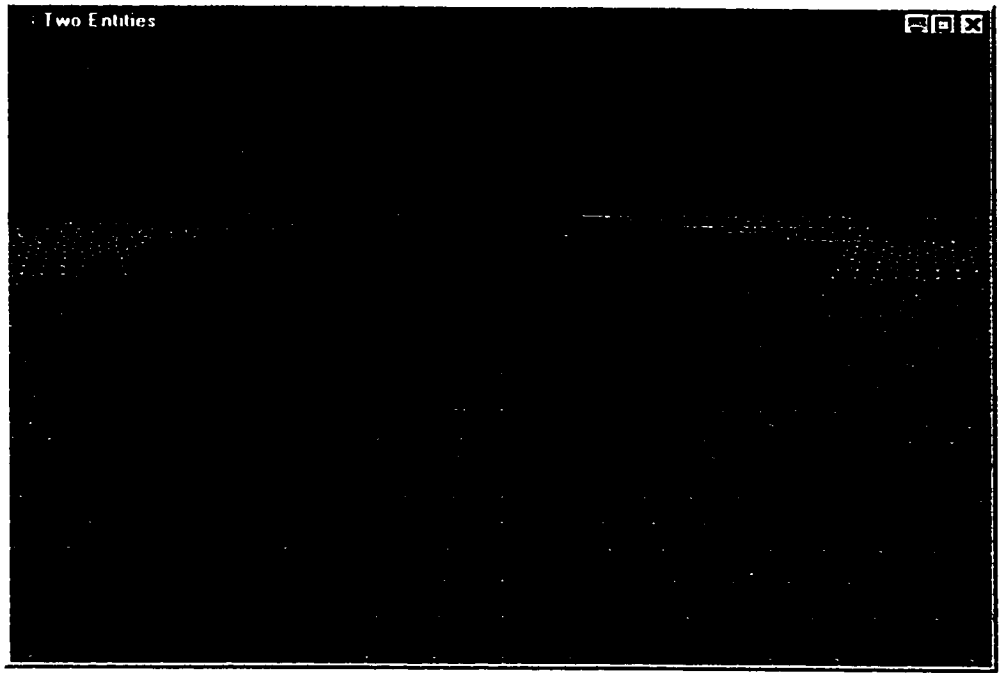

Figure 5-14 Small cube drop down to the table

4.4 Example Four

This example will present a simulation of two sphere objects. It will show a series of activities including objects rotation, movement, zoom in, zoom out, and dropping from the table. The initial simulation parameters and values are shown in table 4-15.

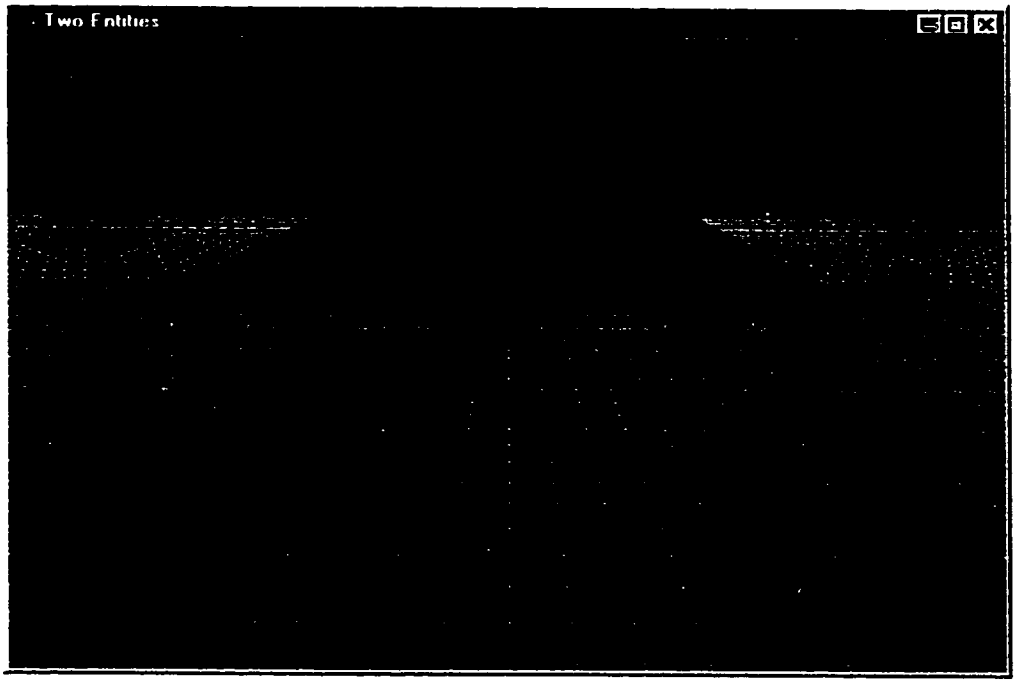

Figure 4-15 Initial positions

Step 1. Start the program

Before simulation, it shows three objects: two spheres and a table. The two spheres are on the table.

Step 2. Set the position and direction of a force

Select menu 'Push Direction', an arrow will appear onto a sphere; press Space key to adjust the arrow direction and position.

Step 3. Start simulation

- Select menu 'Rotation', the spheres and table will rotate a specified angle. See figure 4-16.
- Click on menu 'Zoom In / Zoom Out'', objects will enlarge or decrease. See figure 4-17.
- Click 'Reset', all objects will return to initial positions.
- Press 'G' key, the sphere with force will scroll a distance toward the arrow direction. See figure 4-18.
- Whenever the collision happens, both spheres will move toward the different trajectories. Once a sphere passes over the border of table, it will drop down to ground. See figure 4-19.

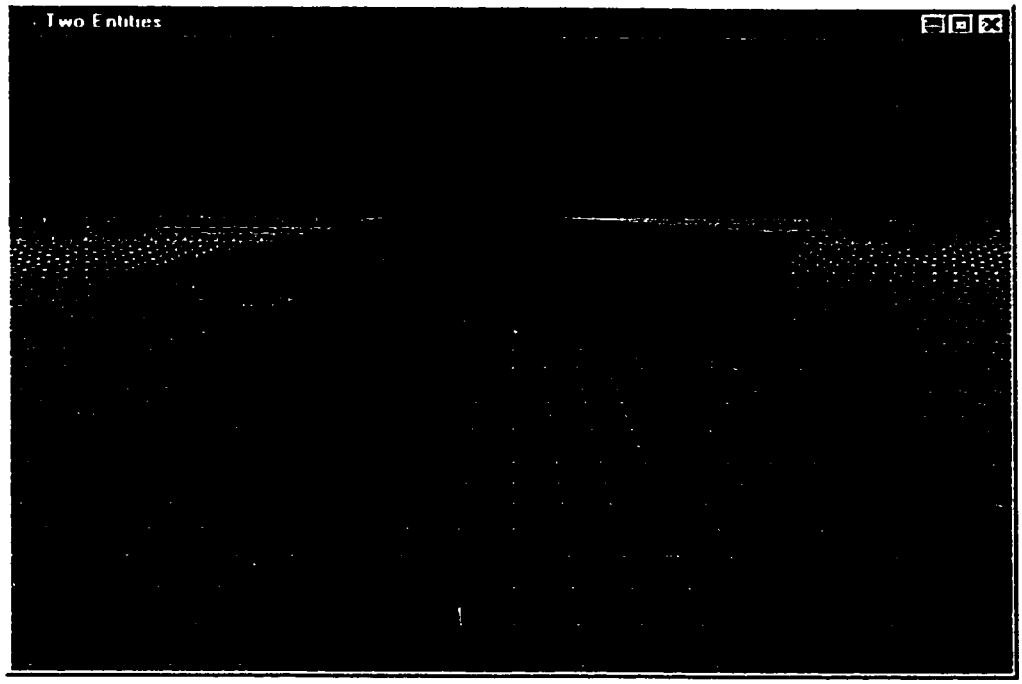

Figure 4-16 Spheres and table rotate a specified angle

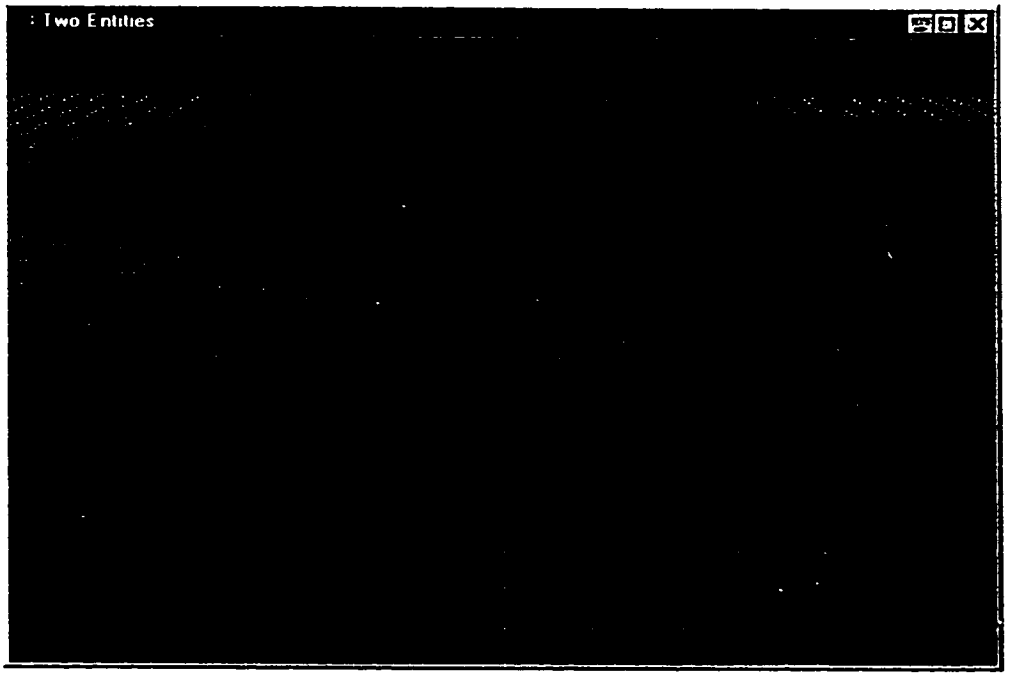

Figure 4-17 Objects enlarge

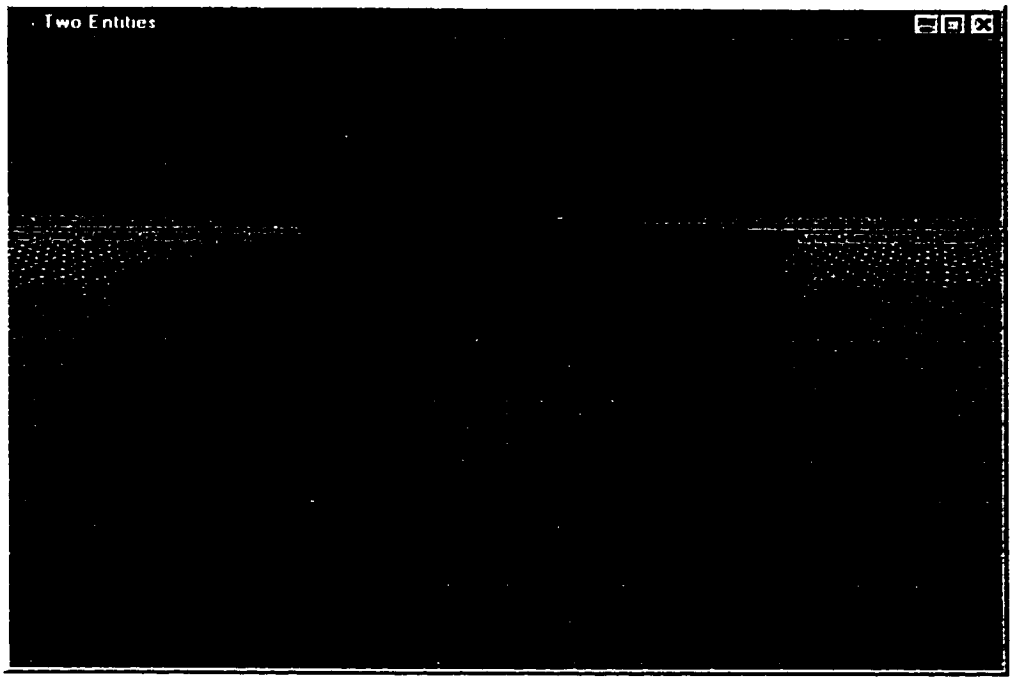

Figure 4-18 sphere scrolls a distance toward the arrow direction

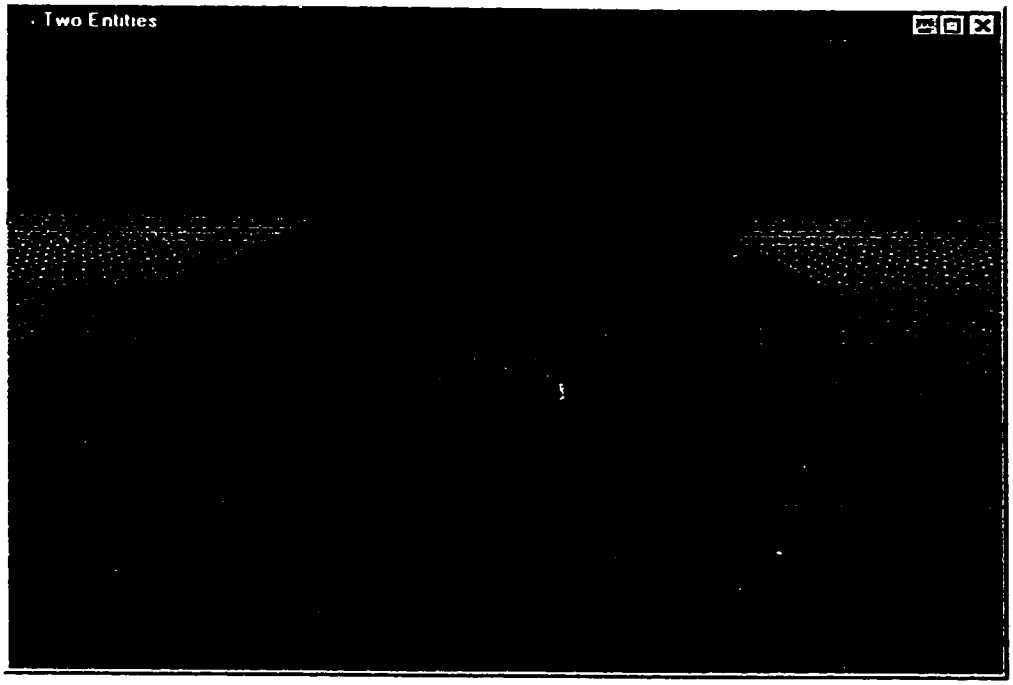

Figure 4-19 Sphere drops down from the table

5. Conclusion

The four examples above demonstrate 3D object simulations. They offer a convenient and efficient way to observe the interactions among objects. The results are correct and interesting.

5.1 Experiences on Object-Oriented Programming

Traditional programming languages separated data from functionality. Typically, data was aggregated into structures that then were passed among various functions that created, read, altered, and otherwise managed that data.

C++, as an object-oriented language, is concerned with the creation, management, and manipulation of objects. An object encapsulates data and methods used to manipulate the data.

Object-oriented programming offers a new and powerful model for writing computer software. It speeds the development of new programs, improves the maintenance, reusability, and modifiability of software. Object-oriented programming focuses on the creation and manipulation of objects, such as cubes, spheres. This type of programming gives us a greater level of abstraction; we can concentrate on how the objects interact without having to focus on the details of the implementation of the object.

5.2 Further Work

One of the limitations of the system is that it does not add enough physical performance in class abstraction. Therefore we suggest the following future work:

- Adding physics modeling, as shown in figure 5-1.
- More powerful games framework.

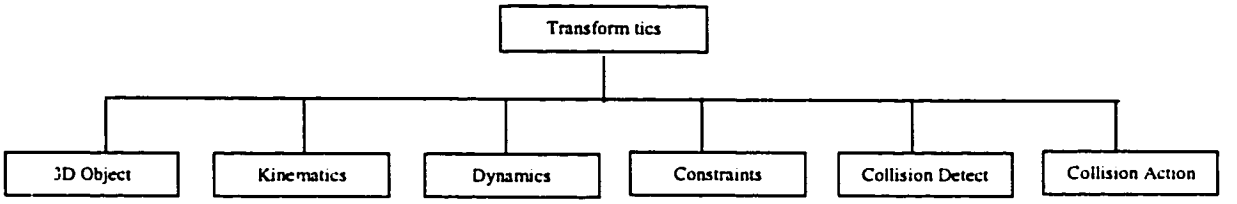

Figure 5-1. Future physics modeling

Bibliography

[Pg98] Peter Grogono. Getting Started with OpenGL. Concordia University, 1998

[JMWFW97] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. Object-Oriented Modeling and Design. General Electric Research and Development Center Schenectady, New York, 1997

[Rsw96] Richard S. Wright JR. OpenGL Super Bible, 1996

[MJT99] Mason Woo, Jackie Neider, and Tom Davis. OpenGL Programming Guide. Third Edition. Addison-Wesley, 1999

[Hs98] Herbert Schildt. C++: The Complete Reference. Third Edition., 1998

[Kg97] Kate Gregory. Special Edition Using Visual C++ 5, 1997.

[Web3D] 3D world simulation. http://www.martinb.com/